WHOLE-SYSTEM TESTING AND ANALYSIS OF ACTOR PROGRAMS

BY

SIHAN LI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Professor Gul Agha, Chair
Professor Darko Marinov
Professor Madhusudan Parthasarathy
Professor Yuanyuan Zhou, University of California, San Diego

ABSTRACT

As multi-core processors and networked systems become the norm, concurrent programming has been widely adopted in industry. Practitioners have used concurrent programming models to build many complex and large-scale systems and infrastructures. With the growing popularity of concurrent programming, it is important to assure the reliability of concurrent systems. However, concurrent systems are notoriously difficult to understand, test, and debug, because the interleaving between concurrent processes leads to non-deterministic behaviors in the systems. The number of non-deterministic behaviors grows exponentially, making it challenging to test and analyze concurrent programs at the system level.

In this dissertation, we target the problem of system testing and analysis of concurrent programs. We first present a characteristic study on real-world bugs in distributed data-processing production systems to identify common challenges and opportunities in reliability assurance of generic concurrent systems, and motivate the need of testing and analyzing the systems as a whole. Then we describe two solutions to the problem in the context of the Actor model, a popular concurrent programming model based on asynchronous message passing. Actors facilitate building scalable systems by making unintended race conditions and deadlocks less likely. Many large systems such as Twitter, LinkedIn, and Facebook Chat, as well as frameworks such as Microsoft Orleans have used the Actor model. In particular, we propose a target test generation method for effective testing of actor systems, and a behavioral specification inference method for understanding and analyzing actor systems.

We conduct a comprehensive characteristic study on 200 production failures and their fixes in a distributed data processing system from Microsoft Bing. We investigate not only major failure types, failure sources, and fixes, but also the debugging practice. Our main findings include (1) one major type of failures is caused by defects in data processing due to frequent data schema changes and exceptional data; (2) another major type of failures is due to the non-determinism in the interactions between a group of concurrent processes; (3) detecting and diagnosing these bugs often requires system level testing and analysis, because in many cases, the root cause of the failure lies in a different process from the failure-manifesting process. Although we study bugs in only distributed data-processing systems, we believe that the second and third findings can be extended to other concurrent systems based on different concurrency models.

Motivated by this study, we develop automated solutions to help practitioners improve the reliability of actor systems. To facilitate system testing, we propose a method to support

generation of system-level tests to cover a given code location in an actor program. The test generation method consists of two phases. First, static analysis is used to construct an abstraction of an entire actor system in terms of a *message flow graph* (MFG). An MFG captures potential actor interactions that are defined in a program. Second, a *backwards symbolic execution* (BSE) from a target location to an "entry point" of the actor system is performed. BSE uses the MFG constructed in the first phase of our targeted test generation method to guide the execution across actors. Because concurrency leads to a huge search space which can potentially be explored through BSE, we prune the search space by using two heuristics combined with a feedback-directed technique. We implement our method in TAP, a tool for Java *Akka* programs, and evaluate TAP on the *Savina* benchmarks as well as four open source projects. Our evaluation shows that the TAP achieves a relatively high target coverage (78% on 1,000 targets) and detects six previously unreported bugs in the subjects.

To help understand and reason about actor systems, we propose a method for inferring the specification diagram of an actor system from its implementation. The actor specification diagram is a formal model that rigorously describes the global behaviors of a group of actors, in terms of the type and the number of messages exchanged between actors as well as the temporal order between message sending and receiving events. Our inference method first uses static analysis to infer an abstract specification diagram, which soundly captures all potential message flows and faithfully reflects the temporal orders between events enforced by control flows and message flows. Then our method uses dynamic analysis to detect likely invariants from execution traces of the system to further refine the abstract specification diagram. The refinements include instantiating the number of loop iterations, removing false positives, and discovering additional temporal orders between events enforced through coordination constraints in the system. We implement the inference method in a tool ASpec, and evaluate ASpec on the *Savina* benchmarks as well as two real-world protocols TCP and SIP. The evaluation results show that ASpec is effective in inferring the actor specification diagrams with high accuracy (78 %) on the subjects.

To my wife and parents, for their love and support.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Professor Gul Agha for his advice and guidance during my Ph.D. study. He taught me how to be an independent researcher to overcome challenges and obstacles in research. He also gave valuable feedback on my research proposals to keep me in the right direction, and provide insights in our technical discussions. I am very grateful for the support and the resources that he offered for me to accomplish my Ph.D. study. I have been so fortunate to be his student.

Besides my advisor, my sincere gratitude goes to the rest of my thesis committee members, Professor Darko Marinov, Professor Madhusudan Parthasarathy, and Professor Yuanyuan Zhou, for their valuable questions and comments in my exams as well as their insightful feedback and suggestions on my thesis. I am also thankful for my colleagues, collaborators, and fellow labmates, Xusheng Xiao, Wei Yang, Farah Hariri, Minas Charalambides, Atul Sandur, Dan Plyukhin, Si Liu, Qi Wang, and Liyi Li etc., who have been very friendly and supportive in my Ph.D. life. I should also thank Krishna Kura for his help on transforming the benchmarks to Java Akka and thank Nadeem Jamali for his helpful comments and suggestions to the thesis material.

I am also grateful for the funds that supported my Ph.D. study. The work in this thesis is supported in part by the National Science Foundation under grants NSF CCF-0845272, NSF CCF-0915400, NSF CCF 14-38982, and NSF CCF 16-17401.

Last but not least, I would like to thank my family, especially my wife and my parents, for their endless love and support. They kept me motivated, gave me confidence, and helped me through difficulties in my Ph.D study. My deepest gratitude goes to my wife for her companionship and her tremendous love. We were in a long-distance relationship for 5 years, and faced numerous difficulties together. Finally we made it, and we live together now.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The radical advance of computing platforms such as clouds and multi-cores has disrupted the field of computer programming. Mainstream programming in industry is shifting from sequential programming models to concurrent programming models. Practitioners have adopted various concurrency models to build complex and large-scale systems and infrastructures. Successful examples of such adoption include the Actor model [1, 2, 3] in message passing systems, Remote Procedure Call [4, 5, 6] in micro service architectures, and MapReduce [7, 8] in distributed data processing systems.

With the growing popularity of concurrent programming, reliability assurance of concurrent systems becomes more and more important. Concurrent systems are known to be notoriously difficult to understand, test, and debug, because the interleaving between concurrent processes leads to non-deterministic behaviors of the systems. The number of non-deterministic behaviors grows exponentially, making it challenging to test and analyze concurrent programs at the system level.

In this dissertation, we aim to discover common reliability issues of concurrent systems and identify opportunities to improve the reliability through a comprehensive study of bugs and fixes in real-world concurrent systems. Motivated by this study, we develop automated tools and techniques to effectively test and analyze concurrent systems.

We pick the Actor model as the underlying concurrency model in this dissertation, because it is a promising concurrency model, and has attracted an increasing amount of interests from researchers as well as practitioners. Many actor-based languages have been developed, including Rosette [9], Hal [10], Thal [11], Erlang [12], SALSA [13], E language [14], Ptolemy [15], and Axum [16]. Among them, Erlang has been widely used in industrial projects [17] such as AXD301 ATM switch, Facebook chat backend. In addition, a number of actor frameworks and libraries are developed to allow actor-style programming in popular sequential languages such as Scala (Scala Akka [18], Lift [19]), Java (Java Akka [20], Jetlang [21], actorFoundry [22], GPars [23]), C/C++ (Act++ [24]), Smalltalk (Actalk [25]), Python (Stack-less Python [26], Stage [27]), .NET (Microsoft's Asynchronous Agents Library [28], Retlang [29]), Microsoft Orleans [30].

Actors are concurrent, autonomous entities that communicate via purely message exchanges. Messages are immutable and exchanged asynchronously. Each actor encapsulates its local state and reacts only when receiving a message. Upon receiving a message, an actor can create new actors, send messages to other actors, and perform local computations that may change its own states. Such model makes concurrent programming (1) less error-prone:

because it prevents concurrent memory access issues by isolating the state of each actor, (2) easier to understand by tracking the message flows, and (3) more scalable and portable: because actors have no shared state, and thus can be easily composed together or moved across network without code modification.

1.1 THESIS STATEMENT

This dissertation argues that although challenging, it is necessary to test and analyze concurrent programs at the system level. The dissertation further demonstrates that automated system testing and analysis techniques may be helpful in detecting concurrency bugs and understanding behaviors of concurrent systems.

To support the thesis statement, we conduct a comprehensive study on representative production bugs and their fixes in a distributed data processing system called SCOPE from Microsoft Bing. Our study findings show that a major type of bugs is caused by the non-determinism in the interactions between a group of concurrent processes, and in many cases, the root cause of the bug lies in a process that is different from the bug-manifesting process. This type of bugs is often difficult to reveal and fix by analyzing each concurrent process individually. A holistic analysis of the system is needed. We believe that these findings are not only limited to distributed data-processing programs, but also are general to other concurrent systems

To tackle the issues discovered in our study, we propose a couple of automated techniques for system-level testing and analysis of actor systems specifically. Although our techniques are developed for actor systems, they can be extended to other concurrent systems based on message passing. To expose concurrency bugs in early stage, we propose a targeted test generation method, which generates system-level tests to cover particular code locations in actor systems. Our evaluation shows that the method is effective in covering target code locations as well as detecting concurrency bugs. Moreover, to help developers better understand the system behaviors holistically, we propose an automated method for inferring actor specification diagram, which is a formal behavioral model specifying interactions between actors. Our evaluation shows that the inference method is capable of deriving accurate specification diagrams from system implementations.

1.1.1 Study of Bugs in Concurrent Systems

Microsoft Bing has an internal platform called SCOPE [31] for processing big data. The platform is built atop the programming model Dryad [32], which is designed for writing

distributed data-parallel programs. Inside Microsoft, SCOPE has been adopted by thousands of developers from tens of different product teams for index building, web-scale data mining, search ranking, advertisement display, etc. Currently, there are thousands of SCOPE jobs per day being executed on Microsoft clusters with tens of thousands of machines. A notable percentage of these jobs threw runtime exceptions and were aborted as failures. Among all failures from Microsoft clusters within two weeks, 6.5% of them had execution time longer than 1 hour; the longest one executed for 13.6 hours and then failed due to un-handled null values.

A situation similar to the one described above was reported by Kavulya *et al.* [33]. They analyzed 10 months of Hadoop [34] logs from the M45 supercomputing cluster [35], which Yahoo! made freely available to selected universities. They indicated that about 2.4% of Hadoop jobs failed, and 90.0% of failed jobs were aborted within 35 minutes while there was a job with a maximum failure latency of 4.3 days due to a copy failure in a single reduce task. Such failures, especially those with long execution time, resulted in a tremendous waste of shared resources on clusters, including storage, CPU, and network I/O. Thus, reducing failures would save significant resources.

Unfortunately, there is little previous work that studies failures of data-parallel programs. The earlier-mentioned work by Kavulya *et al.* [33] studies failures in Hadoop, an implementation of MapReduce. Their subjects are Hadoop jobs created by university research groups, being different from production jobs. Besides, state-of-the-art data-parallel programs in industry are written in hybrid languages with a different and more advanced programming model than MapReduce. Thus, their results may not generalize to these industry programs. Moreover, they focus on studying the workloads of running jobs for achieving better performance by job scheduling, rather than failure reduction in development.

To fill such a significant gap in the literature and the academia/industry, we conduct the first comprehensive characteristic study on failures and fixes of state-of-the-art production data-parallel programs for the purpose of failure reduction and fixing in future development. Our study includes 200 SCOPE failures/fixes and 50 SCOPE failures with debugging statistics randomly sampled from Microsoft Bing. We investigate not only major failure types, failure sources, and fixes, but also current debugging practice. Note that our study focuses on only failures caused by defects in data-parallel programs, and excludes the underlying system or hardware failures. Particularly, the failures in our study are runtime exceptions that terminate the job execution. We do not study failures where the execution is successfully finished but the produced results are wrong, since we do not have test oracles for result validation. Moreover, all SCOPE jobs in our study are obtained from Microsoft clusters. Programmers may conduct local testing before they submit a job to clusters. Hence, some

failures that are addressed by local testing may not be present in our study.

Specifically, our study intends to address the following research questions:

RQ1: *What are common failures of production distributed data-parallel programs? What are the root causes?* Knowing common failures and their root causes would help programmers to avoid such failures in future development.

RQ2: *How do programmers fix these failures? Are there any fix patterns?* Fix patterns could provide suggestions to programmers for failure fixing.

RQ3: *What is the current debugging practice? Is it efficient?* Efficient debugging support would be beneficial for failure fixing.

In summary, the study has the following findings and implications. The most significant characteristic of failures in data-parallel programs is that most failures (84.5%) are caused by defects in data processing rather than defects in code logic. The tremendous data volume and various dynamic data sources make data processing error-prone. More knowledge on data properties, such as nullable for a certain column, could help improve code reliability. We also find that most failures can be put into a few categories, and there are limited corresponding failure sources (e.g., exceptional data) and fix patterns (e.g., setting default values for null columns). Such knowledge on failure types, failure sources, and fix patterns would bring substantial benefits to failure reduction and fixing. Moreover, to balance the cost of data storage and shifting, the SCOPE debugging tool used in practice enables locally debugging only the computation stage where the failure is exposed (i.e., failure-exposing stage). It does not work well if the root cause of the failure is not at the failure-exposing stage. This finding implies that whole-program debugging with low cost is needed in some cases. Although our study is conducted on only the SCOPE platform, we believe that most of our findings and implications can also be generalized to other similar data-parallel systems.

Besides our study results, we also share the current practices and ongoing efforts on failure reduction and fixing. There is a series of tools that could be used to improve reliability of SCOPE jobs, including compile-time checking, local testing, failure reproduction, and fix suggestion.

1.1.2 Targeted Test Generation

We propose a method for generating targeted tests for actor systems based on *backward symbolic execution* (BSE). The tests we generate are system-level tests: they exercise a group of interacting actors rather than only an isolated actor. The goal is to find if a particular line can be reached through sending messages to the *entry point* of an actor system, where an entry point is a message handler of an actor which interacts with the external environment.

In actor terminology, such actors are called *receptionists*. Each test consists not only of the messages received by each actor but also the order in which these messages are received. We start a BSE from the target code and explore only those paths that are relevant to reaching that target; the exploration continues until a feasible path to an entry point of the actor system is found.

In sequential programs, a call graph is used to guide the inter-procedural BSE [36, 37, 38]. In the actor context, we propose to use an abstraction of an actor system called *message flow graph* (MFG). An MFG captures interactions between actors and is useful to guide inter-actor BSE. We develop a sound whole-system static analysis to construct MFGs for actor systems.

One challenge in static MFG construction and BSE for actor systems is to handle actor operations such as message send/receive and actor creation. Even when an actor framework is written in a language like Java, analyses that treat these actor operations as normal methods would not work: if the actor semantics is ignored, BSE will explore the library methods that are used to implement an actor runtime. Because a library that implements an actor runtime contains complex multi-threading and networking code, symbolic execution would become infeasible (cf. [39]). In addition, a static analysis would not be able to establish connections between actors without understanding the meaning of these library methods. To solve this problem, we define formal semantic models of actor operations in both MFG analysis and BSE, and replace actual implementations of actor operations with the semantic models. Assuming that the actor library has been correctly implemented, we prevent our analysis from exploring the underlying library. This makes our analysis more efficient and thus scalable.

In general, it is computationally intractable to consider every possible message arrival schedule even if we explore only paths that are relevant to a single target. To efficiently navigate the search space, we use a depth-first search strategy combined with two heuristics and a feedback-directed search technique. The depth-first strategy attempts to reach the entry point of the actor system as soon as possible. The two heuristics are as follows:

1. Each message handler is executed atomically so that search space is reduced due to the lack of the interleaved execution of message handlers. This heuristic applies the macro-step semantics in the Actor model [2], which follows from the fact that messages to a given actor are processed one at a time and that actors do not share state.

2. Low weights are assigned to transitions in BSE that introduce more actors to be explored, in order to avoid unnecessary explorations. The heuristic is based on the conjecture that most concurrency bugs may be triggered by considering interactions of

a small number of actors. We do not have direct evidence of this conjecture. However, there is a previous finding that most concurrency bugs in multi-threaded programs may be triggered using only two threads [40].

As constraints are collected and solved, some paths turn out to be infeasible. In this case, we deduce an *unsatisfiable core*–a subset of the constraint clauses whose conjunction is unsatisfiable. Our feedback-directed technique uses these unsatisfiable cores to effectively drive BSE towards a feasible path. The technique is particularly useful in cases where BSE frequently hits infeasible paths.

We have implemented our method in a tool called TAP for the Java *Akka* framework [20], a popular library enabling actor-style programming in Java. However, our method can be applied to other actor frameworks or languages. We evaluate TAP on *Savina* [41], a set of 30 third-party actor benchmarks, as well as on four open source actor projects from GitHub. The evaluation results show that TAP is effective in covering targets, achieving 78% target coverage on a total of 1,000 targets. The heuristics and the feedback-directed technique together substantially improve the target coverage over random search. In addition, TAP detects six previously unreported bugs in the subjects, five of which are crash bugs caused by out-of-order message delivery.

1.1.3 Specification Diagram Inference

As mentioned earlier, the Actor model has been adopted to develop many distributed communication-based systems. A typical way of implementing actor systems is to develop the logic of each actor individually in terms of their behaviors upon receiving various types of messages. While this development paradigm keeps the actor code modular, it can present a significant challenge for developers who need to understand the global behavior of an actor system rather than individual actors. To understand and reason about the behaviors of an actor system, developers need to track the message flows between actors and infer the behavior of each actor triggered by a certain type of messages. Moreover, the concurrency in an actor system leads to nondeterminism in message arrival orders. Such nondeterminism in message schedules may result in nondeterministic system behaviors, making it more difficult to understand the system.

The task of understanding the behaviors of an actor system is challenging yet necessary in practice. Examples of such scenarios include diagnosing unexpected behaviors in a system, maintaining and making changes to legacy code of a system. It is difficult and impractical for developers to manually examine or reason about every possible behavior of the system,

because the number of potential behaviors grows exponentially in the number of messages exchanged. In addition, the detail of the actual implementation is often a distraction for developers to understand the global behaviors at a high level.

To address this problem, we propose an automated method for inferring a behavioral model of an actor system. Specifically, we infer *actor specification diagram* — such diagrams rigorously describe the global behavior of an actor system in terms of actor events such as sending and receiving messages as well as the temporal orders between the events. A specification diagram is more expressive than a message flow graph — it can specify the exact number of messages exchanged as well as the temporal order between messages. Since the inferred model focuses on describing how actors interact with each other, it can be used by developers to obtain an overview of the global system behaviors or to detect bugs at the system level.

The proposed inference method operates in two phases. In the first phase, we use static analysis to construct an abstract specification diagram from the system code. The abstract specification diagram captures the events of potential message exchanges between actors as well as the temporal order between these events. Temporal orders are established from the control flows within individual actors and the causal relationship of message send and receive events between actors. While static analysis constructs the specification diagram in a sound manner (i.e., captures all possible events), the resulting specification diagram may also include behaviors that can never happen in concrete executions (called "false positives"). Furthermore, it is difficult for static analysis to infer the number of iterations of a loop, or to infer the coordination constraints encoded in actor states. Hence, in the second phase, we refine the abstract specification diagram produced in the first phase using dynamic information from concrete execution traces. In particular, we dynamically detect *likely invariants* [42] of the actor system, and use them for refining the abstract specification diagram. We identify common patterns of imprecision in the specification diagram constructed by static analysis, and propose an adhoc yet effective refinement for each pattern. We match the specification diagram to a set of pre-defined patterns, and apply corresponding refinements based on the dynamic invariants. This process continues iteratively until there is no applicable refinement, and then a more precise specification diagram is produced.

We implement the inference method in a tool called ASpec, and evaluate ASpec on the *Savina* benchmarks as well as two real-world protocols TCP and SIP. The evaluation results show that ASpec is effective in inferring the actor specification diagrams with high accuracy. ASpec infers accurate specification diagrams for 25 out of the 32 subjects.

1.2 CONTRIBUTIONS

This dissertation contains the following research contributions:

- We present the first comprehensive characteristic study on failures and fixes of production data-processing programs and provide valuable findings and implications for future development and research.
- We introduce the MFG abstraction for actor systems and develop a sound static analysis to construct it.
- We formally define the full semantics of actor operations for MFG analysis and BSE on actor systems.
- We propose two search heuristics and a feedback-directed technique to efficiently navigate the generally huge search space in BSE of actor systems.
- We propose a hybrid approach combining static analysis with dynamic analysis for inferring accurate specification diagrams of actor systems.

1.3 THESIS ORGANIZATION

The rest of this dissertation is organized as follows. Chapter 2 describes the methodology, findings and implications of the characteristic study on bugs in distributed data-processing programs. Chapter 3 describes the target test generation method, its implementation and evaluation. Chapter 4 describes the inference method for actor specification diagram, its implementation and evaluation. Chapter 5 discusses related work. Chapter 6 makes concluding remarks and proposes future work.

CHAPTER 2: STUDY ON BUGS IN CONCURRENT SYSTEMS

In this chapter, we first give a short introduction on the distributed data processing system SCOPE and its underlying programming model. We then describe our study methodology in detail. Next, we present the findings and their implications, followed by discussions on future research directions and opportunities on reliability assurance of concurrent systems. This chapter is based on our previous work [43].

2.1 SCOPE BACKGROUND

SCOPE is the production data-parallel computation platform for Microsoft Bing services. The SCOPE language is a hybrid of declarative SQL language for expressing high-level data flow and imperative C# language for implementing user-defined functions as local computation extension, similar to Pig Latin [44], Hive [45], FlumeJava [46] and Microsoft DryadLINQ [32]. The rest of this section describes the SCOPE data model and programming model, as well as the execution and life cycle of a SCOPE job.

2.1.1 Relational Data Model

SCOPE provides a relational data model like SQL, which encapsulates data sets with *column*, *row*, and *table*. A table consists of a set of rows; a row consists of a set of columns with primitive or complex user-defined types. Each table is associated with a well-defined schema represented as $(columnName_1 : Type_1, ..., columnName_n : Type_n)$. Columns are accessed by either name or index in the form of $row[columnName]$ or $row[columnIndex]$.

2.1.2 UDF Centric Programming Model

The programming model of SCOPE provides three elementary operators: *processor*, *reducer*, and *combiner*, as the base classes for all user-defined functions (UDFs); while *extractor* and *outputter* derived from *processor* are dedicated to read from and write to underlying data streams. *Processor* and *reducer* are similar to *mapper* and *reducer* in MapReduce, respectively. SCOPE extends MapReduce with *combiner*, which generalizes *join* on heterogenous data sets. SCOPE offers built-in implementations of many common relational operations for programmers' convenience, and also allows programmers to implement customized operators as C# UDFs. Relational operations like *filter*, *selection*, and *projection* are achieved

```
1  public class CopyProcessor : Processor {
2      public Schema Produces(string[] columns,
3                             string[] args,
4                             Schema input_schema) {
5          return input_schema.Clone();
6      }
7      public IEnumerable<Row> Process(RowSet input,
8                                      Row output_row,
9                                      string[] args) {
10         foreach (Row input_row in input.Rows) {
11             input_row.CopyTo(output_row);
12             yield return output_row;
13         }
14     }
15 }
```

Figure 2.1: The simplest user-defined processor `CopyProcessor`, which returns the copy of input. Any UDF processor inherits from `ScopeRuntime.Processor`, and implements two methods: (1) `Produce`, which defines the output schema, (2) `Process`, which implements the processing logic and generates the output.

by *processors*, while *distinct* can be implemented as a *reducer*. Figure 2.1 illustrates a simple *processor*, which sequentially copies every input row. From a structural perspective, SCOPE models the application workflow as a direct acyclic graph (DAG), different from MapReduce, which strictly follows a two-phase workflow with *mapper-reducer*.

2.1.3 Job Execution

A SCOPE job consists of input data, compiled binaries, and a DAG execution plan describing *computation* and *data-shuffling* stages. A computation stage includes one or more chained operators, starts after all its predecessors have finished, and independently runs on a group of machines with partitioned data. A data-shuffling stage then connects two consecutive computation stages by transmitting requisite data among machines. A typical SCOPE job has three phases: *data extraction*, extracting raw data into a structured table format; *data manipulation*, manipulating and analyzing data; and *output*, exporting results to an external storage.

Figure 2.2 shows a sample SCOPE job with its execution plan. An external DLL file is first explicitly referenced (Line 1). Next, rows of typed columns (Line 2) are extracted

```
1  REFERENCE "/my/PScoreReducer.dll";
2  t1 = EXTRACT query:string, clicks:long, market:int,...
3       FROM "/my/click_1029"
4       USING DefaultTextExtractor()
5       HAVING IsValidUrl(url);
6  t2 = REDUCE t1 ON query
7       PRODUCE query, score, mvalue, cvalue
8       USING PScoreReducer("clicks")
9  t3 = PROCESS t2 PRODUCE query, cscore
10      USING SigReportProcessor("cvalue")
11      OUTPUT t3 TO "/my/click/1029";
```

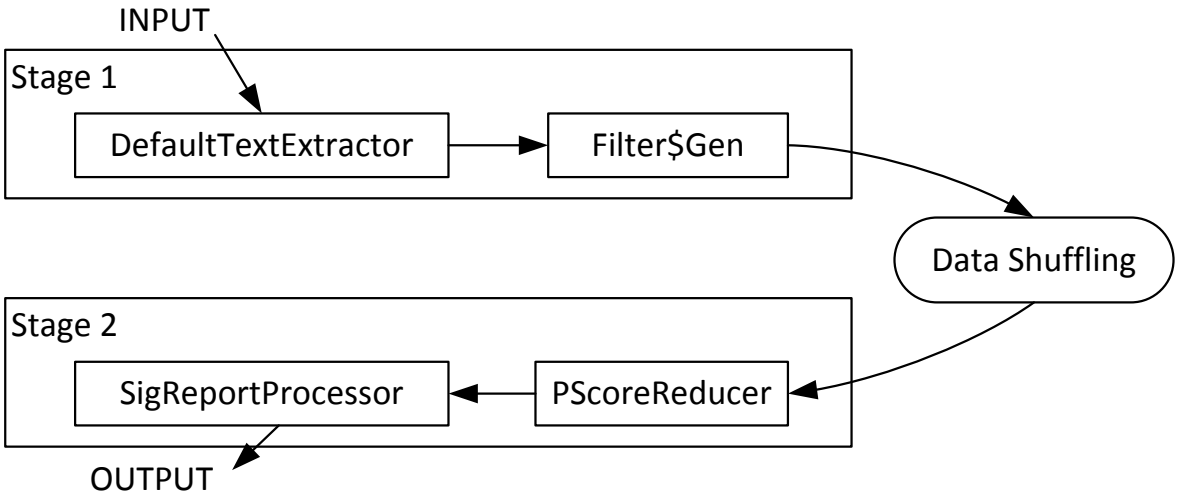


Figure 2.2: A SCOPE job with its execution graph.

from a raw log file (Line 3) as the initial input using a default text extractor (Line 4) and filtered by certain conditions (Line 5). Then input rows are fed to a user-defined operator `PScoreReducer` (Line 8) to produce a new table with four columns (Line 7). Finally, the user-defined operator `SigReportProcessor` (Line 10) is applied and the ultimate result is exported (Line 11). In the execution plan graph, the `Filter$Gen` operator is generated from the `HAVING` clause at Line 5; other operators correspond to keywords `EXTRACT`, `REDUCE`, `PROCESS`, respectively. Each directed edge represents the data flow between operators.

2.1.4 Life Cycle

The typical life cycle of a SCOPE job starts from the development phase. After being tested locally, source scripts are submitted to clusters and the job is executed in parallel. If the job fails or ends with an unexpected result, the programmer downloads relevant data to a local computer and starts debugging in the SCOPE IDE, the environment for developing SCOPE jobs. After defects are fixed, the patched job is re-executed until the correct result is returned.

2.2 METHODOLOGY

2.2.1 Subjects

We took real SCOPE jobs submitted by Microsoft production teams as our study subjects, and collected all job information including initial input data, source scripts, compiled binaries, execution plan, and runtime statistics.

Failures in our study were runtime exceptions that terminated job executions. We did not study failures where the execution was successfully finished but the produced results were wrong since we did not have test oracles for result validation. Hence, the job that ended without exceptions was regarded as a successful one.

Sample Set A. To study failures and fixes in production jobs, we collected all Failed/Successful (F/S) job pairs within two weeks by matching both job names and submitter names. 200 F/S job pairs were randomly sampled out as Sample Set A.

Sample Set B. To study the current debugging practice in SCOPE, we collected all failed jobs that were debugged using the local debugging tool, along with their debugging statistics. We randomly sampled 50 of them as Sample Set B.

2.2.2 Classification and Metrics

Failure classification for Sample Set A was done manually. We first carefully went through all 200 F/S job pairs and understood why the failures happened and how they were fixed. Then we classified these failures from the data point of view: whether the failure was related to input data, and which data level (table or row) triggered the failure. Furthermore, we classified these failures based on their exception types obtained from the error messages.

We next describe the metrics used in our study. First, the number of lines of source code (LOC) was used to measure the size of fixes. We relied on the *WinDiff* tool to find

Table 2.1: Classification of SCOPE failures

Dimension	Category	Count	Ratio
Table Level	Undefined Column	24	12.0%
	Wrong Schema	16	8.0%
	Others	5	2.5%
	Subtotal	45	22.5%
Row Level	Incorrect Row Format	45	22.5%
	Illegal Argument	34	17.0%
	Null Reference	21	10.5%
	User-defined	14	7.0%
	Out of Memory	3	1.5%
	Others	7	3.5%
	Subtotal	124	62%
Data Unrelated	Resource Missing	10	5.0%
	Index Out of Range	9	4.5%
	Key Not Found	5	2.5%
	Others	7	3.5%
	Subtotal	31	15.5%

differences between the failed and successful scripts, and then manually counted the LOC of changes belonging to the fixes since there might be fix-irrelevant code modifications. We also measured the execution time of SCOPE jobs and the size of downloaded data for debugging, which were directly obtained from runtime statistics and SCOPE IDE logs.

2.2.3 Threats To Validity

Threats To Internal Validity. Subjectiveness in the failure classification was inevitable due to the large manual effort involved. Besides, there also might be human mistakes in counting LOC and filtering fix-irrelevant code changes. These threats were mitigated by double-checking all manual work. If there were different opinions, a discussion was brought up to reach an agreement.

Threats To External Validity. We conducted our study within only Microsoft, making it possible that some of our findings might be specific to SCOPE and would not hold in other systems. Hence, we do not intend to draw general conclusions for all distributed data-parallel programs. In next section, we discuss in detail which findings could be generalized to other systems similar to SCOPE.

```
QueryData = SELECT RawQuery, FormCode,
-           Frequency,...
+           Market, Frequency,...
            FROM RawData;
FilteredData = PROCESS QueryData
               USING FormCodeFilter();
...
public class FormCodeFilter: Processor {
    ...
    if(allowedMarkets.Contains(row["Market"].String))
    ...
}
```

Figure 2.3: A real-world example of an undefined-column failure. The `Market` column is not selected into `QueryData` but accessed through expression `row["Market"]`, causing the failure. The fix is adding the `Market` column to `QueryData`.

2.3 FINDINGS AND IMPLICATIONS

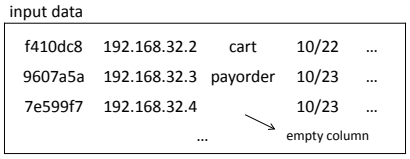
In this section, we first present the failure classification in Sample Set A, and then describe root causes and fixes for each category. Finally, we summarize what we learned from real-world cases.

2.3.1 Failure Classification

Compared to traditional counterparts, data-parallel programs are more data centric and their code logic generally focuses on data analytics and processing. Table 2.1 depicts the classification of 200 failures in Sample Set A. We classify these failures into two big categories: data-related failures and data-unrelated failures. Most failures (169/200) belong to the former as being caused by data-processing defects, while only 31 failures belong to the latter due to defects in code logic, or other reasons. We further divide data-related failures into *table-level* and *row-level*, and build subcategories by failure exception types. We next go through each failure category in detail: how the failure happens, what is the root cause, and how to fix it.

Table-Level Failures

A failure is regarded as *table-level* if every row in the table could trigger it. 22.5% failures in Sample Set A are *table-level* failures, classified into the following major subcategories.




```
  ResultSet = SELECT ClientId, UserIp,
              ScenarioName, Date,...
              FROM @InputData
-             USING DefaultTextExtractor();
+             USING DefaultTextExtractor("-silent");
  ...
```

Figure 2.4: A real-world example of a *row-level* column-number mismatch due to a null column value. The `DefaultTextExtractor` extracts one fewer column for the third row because of the null value for column `ScenarioName`. The failure is fixed by adding the silent option to filter out rows with incorrect format.

Undefined Column. This subcategory is the most frequent *table-level* failures (24/45). SCOPE enables column access by either name or index. Such failures occur when an invalid column is referenced: its column name cannot be found or its index is not within the range. Figure 2.3 shows an undefined-column example. In the C# code, the expression `row["Market"]` gets the value of column named *Market* in the row. This operation is similar to accessing items in a dictionary. However, there is no `Market` column in the `QueryData` table because it is not produced by the `SELECT` statement. The fix, the code in red, is straightforward by adding the `Market` column in selection. An immediate impression from this example is that an undefined column can be detected at compile-time. It is true in this example because the `Market` column is accessed through a constant string name. However, the column name or index could be variables whose values are determined at runtime. In this case, the compiler can never decide whether the column access is valid or not. Hence, column-access validity is always checked at runtime.

Wrong Schema. A wrong-schema failure (16/45) usually occurs in the data-extraction phase, where raw input data are extracted into a structured table for later manipulation. A wrong schema is caused by mismatch of either the column number or column type. For example, there are 10 columns in the input while only 9 columns are defined in the schema, or the first column contains float values while it is declared as integer.

There were two major reasons that led to *table-level* failures. One reason (13/45) was programmers' mistakes. It was common to see misspells in column names and miscounts in column indices. At first, we were a little surprised at the high mistake ratio. However,

```
  ...
  SELECT AVG(a.StrToInt(Duration)) AS AvgD,
        MAX(Helper.StrToInt(Duration)) AS MaxD
        FROM Rowset;
        OUTPUT TO @PARAM_OUTPUT_SET;
  public class Helper {
     public int StrToInt {
        ...
-       if(String.IsNullOrEmpty(val))
+       if(!Int32.TryParse(val, out ivalue))
           ivalue = 0;
-       ivalue = Int32.Parse(val);
        ...
     }
  }
```

Figure 2.5: A real-world example of an illegal-argument failure. Any integer value that exceeds the range of `Int32` will trigger the failure. The fix is adding a safer method `TryParse` to guard against all exceptional data.

after further investigation, we found that real-world SCOPE scripts may involve tables with hundreds or even thousands of columns. In this case, manually writing schemas or accessing columns could be error-prone. Another major reason was the frequent changes of input-data schema without updating processing programs. Since the data were usually from multiple dynamic sources such as web contents or program outputs, the programmers might be unaware of the changes of data sources. We found that the input of quite some failed jobs changed its schema frequently. Even worse, it was often the case that the data producer was not the SCOPE job programmer. Fortunately, for most *table-level* failures, our study indicated that programmers could easily locate the defects based on the error message, and fixes were usually straightforward, such as modifying the schema or correcting the corresponding column name or index.

Row-Level Failures

A failure is said to be *row-level* when only a portion of rows in the table could cause the failure while the other rows are processed successfully. In Sample Set A, there are 124 (62.0%) *row-level* failures including the following major subcategories.

Incorrect Row Format. Incorrect row format is the most frequent subcategory (22.5%) among all failures. Similar to the wrong-schema failure, it also happens in the data extraction

phase due to column-number mismatch or column-type mismatch. The difference is that the incorrect-row-format failure is caused by a few rows with exceptional data rather than schema mismatch. Here, by exceptional data, we mean special cases in data under processing. Thus, only those exceptional rows could trigger the failure while the other rows are well processed. Figure 2.4 shows a real-world example of a *row-level* column-number mismatch due to a null column value, from Sample Set A. The input file in the figure is synthesized by us to simulate real input characters. When processing the third row with the empty column `ScenarioName`, the `DefaultTextExtractor` would not know there is a null value for `ScenarioName` column, and thus extracts one fewer column for this row. Hence, an incorrect-row-format exception is thrown. The failure is fixed by adding the silent option, which tells the `DefaultTextExtractor` to discard rows that cannot be extracted into the correct format.

Illegal Argument. The illegal argument is the second most frequent (17.0%) subcategory. Such failures happen when the argument value does not satisfy the requirement of the invoked method (e.g., requiring none-empty/null value or positive integer). Figure 2.5 shows an example of an illegal-argument failure from Sample Set A. Although the programmer already considers exceptional values like null and empty, and replaces them with default value 0. However, there are some other integer values that exceed the range of `Int32` and thus violate the argument requirement of `Int32.Parse`. The fix is using a safer method `TryParse`, which returns true and assigns the parsed result to `ivalue` if the parsing succeeds; returns false if the parsing fails.

Null Reference. A null-reference failure happens when a null value is dereferenced. The null value usually comes from a null column in data under processing rather than an uninitiated object declared in C# code. A typical example is that a string column contains a null string value, and the programmer performs string operations on this column (e.g., `row["StringColumn"].IndexOf("-")`) without nullity checking so that the null value is dereferenced.

Out of Memory. An out-of-memory failure occurs when the programmer attempts to load extremely huge data into memory all at once. Although we find only 3 such failures, this subcategory is very interesting and important. It reveals an essential difference between distributed data-parallel programming and traditional small-scale counterpart: more memory-efficient algorithms should be devised facing unpredictably massive data. Figure 2.6 shows a typical example that the programmer accumulates all input rows in memory so as to conveniently process them after the last occurrence of a certain pattern. The `input` to `MyReducer` is a group of rows with the same key to be reduced. As the row number of the group could be very large, such an attempt, adding all rows of a group into a list, results in memory exhaustion quickly. Fixing an out-of-memory failure is not straightforward since the pro-

```
t2 = REDUCE t1 ON name
     PRODUCE tag, name, seconds, count
     USING MyReducer();
...
public class MyReducer: Reducer {
  List<Row> list = new List<Row>();
  int last = -1; int i;
  foreach (Row row in input.Rows {
    list.Add(row);
    if (row[3].String == "pattern")
      last = list.Count - 1;
  }
  for (i = last + 1; i < list.Count; i++) {
    ...
  }
}
```

Figure 2.6: A simplified real-world example of the out of memory failure. All input rows are accumulated in memory for later global processing. The fix requires a more memory efficient algorithm.

grammer has to come up with a more memory-efficient implementation for the same code logic.

As we can see from the preceded typical examples, most (99/124) of the *row-level* failures are due to exceptional data. However, we should not blame programmers for these failures because the data volume is so large that it is impossible for programmers to know about all exceptional data in advance.

We find two patterns for fixing the failures caused by exceptional data. One is the row-filtering pattern (43/99), which discards exceptional rows. Since there are millions of rows in datasets, a few exceptional rows could be treated as noises and filtered out without really affecting the job results. The other is the default-value pattern (31/99), which replaces the exceptional values with the default value of its type.

Data-unrelated failures

We find 31 failures not to be closely related to data in Sample Set A. Some of them involve language features while the others are due to semantic errors.

Resource Missing. A resource-missing failure occurs when the job cannot find the needed resources (e.g., referenced scripts or external DLLs) for execution. In Sample Set A, the

main reason (8/10) for resource-missing failure is the programmer's neglect of an important SCOPE language feature. That is, if one needs to reference an external resource, she should explicitly import the referenced resource in the script by using SCOPE keyword `RESOURCE` for script files or `REFENRENCE` for DLL files. Only with these keywords, the SCOPE engine would know that the referenced files should be copied to each distributed machine because these distributed machines are independent and they all need a copy of referenced files for execution. Hence, even if one uploads the referenced resources to the cluster without these keywords, the referenced resources would not be copied to each machine. The tricky part is that in the local development environment, these keywords are not required when the referenced resources are in the default project workspace, making such failures not revealed in local testing. All such failures are fixed by adding keywords `RESOURCE`/`REFENRENCE` to import the missing resources.

Other Data-unrelated Failures. The index-out-of-range and key-not-found exceptions are just like those in ordinary C# programs. The index-out-of-range exception is thrown when accessing an element of an array with the index outside the array bound, and the key-not-found exception is thrown when retrieving an element from a collection (e.g., dictionary) with a key that does not exist in the collection. In Sample Set A, there are various reasons for these failures including the programmer's mistakes and defects in algorithms. Due to these various reasons, the fixes are diverse and we do not find any fix pattern for these failures.

2.3.2 Learning From Practice

The major characteristic of SCOPE failures is that most of them are caused by defects in data processing rather than defects in code logic. Essentially, such characteristic is due to the tremendous volume and dynamism of input data. Since the data are extremely large and come from various domains, it is common that there are missing data or certain special case data. It is impossible for programmers to know all these exceptional data before they really run programs against them. Moreover, the data are usually obtained from multiple dynamic sources, such as web contents and program outputs, which may change frequently. It is difficult and undesirable to keep programmers updated with these changes all the time. These challenges are not unique for SCOPE but also exist in other similar platforms for big-data processing.

However, we can somehow alleviate such problems by providing more information on data. For example, for those data with relatively stable schema, the data producer, such as log-file designers, could provide detailed documentation on data schema or some default data

extractors. The programmer is encouraged to look at the content of data to know more about the data before coding. Moreover, the programmer could leverage domain knowledge to infer some data properties, e.g., a certain column is nullable.

Finding 2.1: Most of the failures (84.5%) are caused by defects in data processing rather than defects in code logic. The tremendous data volume and various dynamic data sources make data processing error-prone.
Implication: Documents on data and domain knowledge from data sources could help improve the code reliability. Programmers are encouraged to browse the content of data before coding, if possible.

The *table-level* failures are mainly caused by the programmers' mistakes and frequent changes of data schema. Since the *table-level* failure could typically be triggered by every row in the table, running the job against a small portion of the real input data could effectively detect these failures.

Finding 2.2: 22.5% failures are *table-level*; the major reasons for *table-level* failures are programmers' mistakes and frequent changes of data schema.
Implication: Local testing with a small portion of real input data could effectively detect *table-level* failures.

Most of *row-level* failures are due to exceptional data. Since the exceptional data are unforeseeable, programmers could proactively write exceptional-data-handling code with domain knowledge to help reduce failures, sharing the similar philosophy with the defensive programming.

Finding 2.3: *Row-level* failures are prevalent (62.0%). Most of them are caused by exceptional data. Programmers cannot know all of exceptional data in advance.
Implication: Proactively writing exceptional-data-handling code with domain knowledge could help reduce *row-level* failures.

How to fix the failures is closely related to the reasons that cause the failures. For most failure categories, there exist fix patterns. Fixes under these patterns are very small in terms of LOC. In Sample Set A, 95.0% fixes are within 10 LOC and 87.0% fixes are within 5 LOC; the average size of fixes is 3.5 LOC. In addition, fix patterns such as row filtering, nullity checking, and resources import, usually do not involve program semantics. Hence, based on these patterns, it is possible to automatically generate some fix suggestions to help programmers fix corresponding defects.

Finding 2.4: There exist some fix patterns for many failures. Fixes are typically very small in size, and most of them (93.0%) do not change data processing logic.
Implication: It is possible to automatically generate fix suggestions to programmers.

2.3.3 Debugging in SCOPE

Debugging in distributed systems is challenging. Since the input data are huge, it is prohibitively expensive to re-execute the whole job for step-through diagnosis. To balance the cost of data storage and shifting, SCOPE enables programmers to debug the failure-exposing stage of the job locally. When a failure happens, the SCOPE system locates the commodity machine where the failure happens, and persistently stores the input data (a partition of the whole stage input) on that machine. Later the programmers could download the input data along with executables from the failed machine, and start live diagnosis in their local simulation environment. The downloaded input would guarantee to reproduce the failure because executions on each distributed machine are independent so that the local environment is the same with that on the failed machine.

To investigate the effectiveness of current debugging practice, we studied Sample Set B, consisting of 50 failed jobs that were debugged with the local debugging tool. An important indicator for effectiveness of the debugging tool was whether programmers came up with correct fixes by using this tool. We found that all the 50 failures in Sample Set B were correctly fixed, which, to some extent, demonstrated the effectiveness of the debugging tool, although the debugging tool might not be the only helper to find a fix. Moreover, from the programmer's perspective, we measured how long it took to initiate the debugger (i.e., time to download debugging-required data to the local machine). Our results showed that 35 out of 50 jobs in Sample Set B downloaded less than 1 Gigabytes data, and the average size of downloaded data for each job was 5.3 Gigabytes. With the high-speed internal network, the debugger could be initiated within few minutes in most cases.

Hence, the debugging tool was efficient in most cases, in terms of quickly reproducing partial failure execution locally. However, we found an interesting case in which the debugging tool may not work well. When a failure occurs, the root cause of the failure may not lie in the computation stage that exhibits the failure (failure-exposing stage). The program state may already turn bad long before the bad state is finally exposed. In this case, debugging the failure-exposing stage may not give sufficient information on how the program state turns bad. Hence, an interesting phenomenon happens that the programmer wants to debug earlier successful stages. We did find such request in the SCOPE internal mailing list.

```
...
Statistics = SELECT StartTime, EndTime,
-             (EndTime-StartTime)/60
+             (EndTime-StartTime)/60.0
              AS Duration, ...
              FROM ResultSet;
...
Out = SELECT TaskID, Name, ... ,
      WaitingTime/Duration AS Percentage
      FROM Info;
...
```

Figure 2.7: A real-world example that illustrates the root-cause problem. A divide-by-zero exception is thrown in the second select statement due to zero values in `Duration` while the root cause is in the first select statement from a different computation stage.

However, it is impractical to enable debugging all successful stages because it would require persistently storing all intermediate data between each stage for later downloading, and the cost is unaffordable. Even if we can afford temporarily storing such intermediate data, we are still unable to download all input data for a stage because they are too huge, sometimes even larger than the original input.

We found that there were 4 out of 50 failures with the root cause outside the failure-exposing stage. Figure 2.7 shows one example of them. A divide-by-zero exception is thrown in the second statement due to the zero value in `Duration`. The root cause for the zero value lies in the first select statement. Since `StartTime` and `EndTime` are integers, (`EndTime` - `StartTime`) / 60 is zero when the numerator is less than 60. However, the first statement is not in the failure-exposing stage and will never be debugged with this tool.

Essentially, this root-cause problem is due to the balance act: partial-program (failure-exposing stage) debugging. To achieve low-cost whole-program debugging, we could try to automatically generate small program inputs to reproduce the entire failure execution by leveraging existing symbolic-execution engines [47, 48, 49]. This approach could be complementary to the current debugging approaches.

Finding 2.5: The current debugging practice is efficient in most cases in terms of fast failure reproduction. However, there are some cases (8.0%) where the debugging tool may not work well because the root cause of the failure is not inside the failure-exposing stage.
Implication: Automatically generating smaller program inputs to reproduce the entire failure execution could be complementary to current debugging approaches.

Table 2.2: Our major findings on failure characteristics of real-world data-parallel programs and their implications

Findings on Failures	**Implications**
(1) Most of the failures (84.5%) are caused by defects in data processing rather than defects in code logic. The tremendous data volume and various dynamic data sources make data processing error-prone.	Documents on data and domain knowledge from data sources could help improve the code reliability. Programmers are encouraged to browse the content of data before coding, if possible.
(2) 22.5% failures are *table-level*; the major reasons for *table-level* failures are programmers' mistakes and frequent changes of data schema.	Local testing with a small portion of real input data could effectively detect *table-level* failures.
(3) *Row-level* failures are prevalent (62.0%). Most of them are caused by exceptional data. Programmers cannot know all of exceptional data in advance.	Proactively writing exceptional-data-handling code with domain knowledge could help reduce *row-level* failures.
Findings on Fixes	**Implications**
(4) There exist some fix patterns for many failures. Fixes are typically very small in size, and most of them (93.0%) do not change data processing logic.	It is possible to automatically generate fix suggestions to programmers.
Findings on Debugging	**Implications**
(5) There are cases (8.0%) where the current debugging tool in SCOPE may not work well because the root cause of the failure is not at the failure-exposing stage.	Automatically generating program inputs to reproduce the entire failure execution could be a complementary approach to current debugging practices.

Summary of Study Results

Table 2.2 summarizes the findings and their implications from our study. We have studied 250 failures of production SCOPE jobs, examining not only the failure types, failure sources, and fixes, but also current debugging practice. The major failure characteristic of data-parallel programs is that most of the failures (84.5%) are caused by defects in data processing rather than defects in code logic. The tremendous data volume and various dynamic data sources made data processing error-prone. In addition, there are limited major failure sources, with existing fix patterns for them, such as setting default values for null columns. We also have revealed some interesting cases where the current SCOPE debugging tool does not work well and provided our suggestions for improvement.

```
#IF (LOCAL)
    #DECLARE input string = "input.tsv";
#ELSE
    #DECLARE input string = "/my/input.tsv";
#ENDIF
```

Figure 2.8: Adaptive data selection by local versus remote execution.

2.3.4 Discussions

In this section, we present a series of our current practices on failure reduction for SCOPE jobs, including language extension of SCOPE and compile-time analysis. Data synthesis and bug-fix suggestion remain as the future work to reduce debugging efforts.
SCOPE Extension. We extend SCOPE with the following embedded language supports.

1. *Nullable Type* is used to tolerate Null-Reference failures. "Nullable" data types defined in C# [50] can be assigned null to value types such as numeric and boolean types. It is particularly useful when dealing with databases containing elements that may not be assigned a value. We extend SCOPE by supporting C# nullable data types, and annotate them with "?" postfix, a shorthand for "Nullable<T>". If a nullable column is null, it returns the default value for the underlying type.

2. *Structured Stream* is designed to avoid failures occurring in the data-extraction phase. It provides schema metadata for unstructured streams so that SCOPE can directly read from and write to structured streams without extractor and outputter, and thus reduces data-extraction failures.

3. *Local Testing* greatly facilitates testing and diagnosis by enabling jobs to run entirely on a single machine with local test inputs. The local execution has nearly identical behaviors to its distributed execution in clusters. If the programmer specifies the *LOCAL* keyword together with the *#IF* directive in the script (see Figure 2.8), the SCOPE IDE generates a special job that has no data-shuffling stages and spawns very few instances. The programmers can test and debug the job step-by-step, as if the job is a local process.

Compile-Time Program Analysis. We have built the SCA (SCOPE Code Analysis) tool, which has been integrated into the SCOPE IDE, to report potential defects before job execution. SCA, built atop FxCop [51] and Phoenix [52] compiler, includes 11 SCOPE-related checking rules, e.g., the null reference and column assignment should accept correct types. A rule is a piece of C# code targeting at a specific failure category. Currently,

SCA is capable of detecting the Undefined-Column, Wrong-Schema, Resource-Missing, and Null-Reference failures. However, there are non-trivial false alarms produced by SCA. Our ongoing efforts focus on reducing these false alarms by performing whole-program analysis across function calls, operators, and stages.

Data Synthesis. Motivated by Finding 2.5, it is useful to develop a small-scale-data synthesis tool for both testing and debugging purposes. First, to help failure diagnosis, we could leverage part of initial inputs and temporary results of the failed job to synthesize much smaller failure-triggering inputs for quick reproduction of the same failure. Second, we could generate special input to trigger the hidden defects to cause failures. It is feasible to implement these features by extending current symbolic-execution techniques.

Fix Suggestion. Motivated by Finding 2.4, it is promising to develop a tool to generate fix suggestions interactively with the programmers. A straightforward implementation is to first identify the failure pattern (type and reason) from error messages and call stacks, and then generate suggestions based on the corresponding fix patterns found in our study.

2.3.5 Generality of Our Study

Although our study is conducted exclusively on SCOPE jobs from Microsoft Bing, most of our results can still be generalized to other data-parallel systems, such as Pig Latin, Hive, and FlumeJava.

From the input-data perspective, the volume of datasets processed by SCOPE jobs ranges from a few gigabytes to tens of petabytes, representing the typical data volume of current big-data applications in industry. Moreover, the data processed by SCOPE jobs are from similar sources (e.g., websites, user logs) with those from web companies, such as Google, Facebook, and Yahoo!. Finally, the relational data model used in SCOPE is widely adopted by data-parallel platforms. From the programming-model perspective, SCOPE shares the same hybrid programming model with Pig Latin, Hive, FlumeJava. Such model is state-of-the-art for data-parallel programming.

Hence, our Findings 2.1, 2.2, and 2.3 about failure characteristics could be generalized, at least to those web companies who share similar data and programming models. For Finding 2.4, we believe that some of our fix patterns, such row filter and default value, also exist in other systems because they are quite intuitive and straightforward ways to handle exceptional data. While other patterns related to SCOPE language features, like adding `RESOURCE` keyword to import resources, are specific in SCOPE. Finding 2.5 can be generalized to systems that enable only partial-program debugging.

2.3.6 System Design for Failure Resilience

In our study, we found a job that executed for 13.6 hours and failed due to a null column value. The defect was easy to locate, and was fixed by just filtering out rows with null column values. Unfortunately, the patched job had to start all over again and execute for another long time.

Motivated by this observation, we propose a stop/resume mechanism for failure resilience. Instead of killing the job right upon a failure, SCOPE job manager could first suspend the job execution, and then notify the programmer of the failure, wait for her to fix the defect. After the programmer submits the patched job, it is re-compiled into a new execution plan, and the job manager resumes the execution by determining the stages needed to be re-computed based on the execution-plan changes. In this manner, we reuse the previously computed results so that resources for re-computation are saved and the job latency is also reduced, compared to re-executing the patched job from the beginning.

One key fact making this stop/resume mechanism promising is that in our study, 93.0% fixes do not change the code logic. This fact implies that a large portion of previous results could be reused. There may be some cases where our proposed mechanism would not work. Example cases are when the fix changes the program a lot or the programmer can not come up with a fix in short time. In such cases, the failure resilience could be turned off by programmers.

CHAPTER 3: TARGETED TEST GENERATION FOR ACTOR SYSTEMS

A goal of testing programs is to detect violations of desired safety properties. Some safety properties such as no "dangling links" or "division by zero" are implicit. Others are explicitly stated in the form of assertions. Violations of safety properties happen if particular lines of the code can be reached with problematic data. Because concurrency leads to nondeterminism, figuring out if particular lines of the code can be reached is challenging. By taking advantage of the actor semantics, more effective testing tools may be developed. One approach [53, 39] is to combine *concolic testing* [49] with *partial order reduction* based on a macro-step actor semantics [2]. Unfortunately, given the very large number of potential message schedules in an actor system, concolic testing is sometimes ineffective in determining if a particular code location can be reached.

An alternate approach is to use a *targeted test generation* technique to try to generate tests that cover specific code locations.[1] Targeted test generation has the advantage that one does not explore paths leading to code locations that obviously cannot have problems. Previous research has developed techniques and tools based on symbolic execution for targeted test generation for sequential programs (e.g., [54, 55, 56, 57, 58, 36, 37, 38]).

In this chapter, we propose a method for generating targeted tests for actor systems based on backward symbolic execution. The tests we generate are system-level test: they exercise a group of interacting actors rather than only an isolated actor. The goal is to find if a particular line can be reached through sending messages to the *entry point* of an actor system, where an entry point is a message handler of an actor which interacts with the external environment.

The rest of this chapter is organized as follows. We first provide background on the Actor model and the Java *Akka* framework, and then describe the targeted test generation problem for actor systems in terms of the inputs and outputs. Next we formally describe our targeted test generation method and its implementation. Finally, we present the evaluation results of applying our technique to benchmarks and real-world open source projects.

[1] Targeted test generation is sometimes called *directed* or *guided* test generation in the literature.

```
1  public class Main {
2    public static void main(String[] args) {
3      ActorSystem system = ActorSystem.create("Banking");
4      ActorRef serverActor = system.actorOf(Server.props()));
5      ActorRef clientActor = system.actorOf(Client.props(serverActor));
6    }
7  }
8  public class Client extends UntypedActor {
9      private double balance = 100;
10     private ActorRef server;
11     @Override
12     public void onReceive(Object message) {
13         if (message instanceof WithdrawMessage) {
14             double amount = ((WithdrawMessage) message).amount;
15             if(balance >= amount) {
16                 balance -= amount;
17                 server.tell(message);
18             }
19         } else if(message instanceof DepositMessage) {
20             double amount = ((DepositMessage) message).amount;
21             balance += amount;
22             server.tell(mesage);
23         }
24     }
25 }
26 public class Server extends UntypedActor {
27     private double balance = 100;
28     @Override
29     public void onReceive(Object message) {
30         if (message instanceof WithdrawMessage) {
31             double amount = ((WithdrawMessage) message).amount;
32             assert(balance >= amount);
33             balance -=amount;
34         } else if(message instanceof DepositMessage) {
35             double amount = ((DepositMessage) message).amount;
36             balance += amount;
37         }
38     }
39 }
```

Figure 3.1: Simplified Bank Account Example

3.1 PRELIMINARIES

3.1.1 The Actor model

In the Actor model [59, 1, 2], an actor is an agent of computation; it performs computations as a response to a message. An actor is characterized by an actor name, a local state, and behaviors. The actor name serves as the address of the actor in the system; it can be passed around to other actors so that they may send messages to it. The local state of an actor is encapsulated within the actor – no external entity can change it directly. The only way to change the local state of an actor is to send it a message that triggers this actor to change its own state. Upon receiving a message, an actor can have the following three behaviors: (1) performing local computations (updating its local state), (2) sending messages to actors, or (3) creating new actors. Communication between actors is through *asynchronous* message passing – the sender does not block its computation waiting on the recipient to process the message, nor does it assume the order in which the recipient processes its incoming messages. Messages are immutable and processed by the recipient one at a time without interleaving. An actor system contains a group of actors. The subset of actors that can communicate with the external environment are called *receptionists*, and the other actors in the system are called internal actors.

3.1.2 Actors in *Akka*

Akka is a set of libraries for developing distributed and scalable systems on the Java Virtual Machine. It can be used in both Scala and Java. The core of *Akka* is the `akka-actor` library, which is an implementation of the Actor model. Figure 3.1 shows a simplified Bank Account example written using Java *Akka*. We use this example to illustrate important concepts of the Actor model in the context of *Akka*.

Actor Creation. To create actors, we need to first create the enclosing actor system (Line 3 in Figure 3.1), a container in which the actors run. Then we create actors that live in the system via the method `actorOf`. The example creates two actors: a client and a server (Lines 4-5). The `actorOf` method takes as input a configuration object (`props`) that specifies the options for creating an actor such as its type and arguments to its constructor, and returns an `ActorRef` object, which represent the address of the actor in the system. The `ActorRef` corresponds to the concept, actor name in the Actor model. Following the naming convention in *Akka*, we will use the terms actor reference and actor name interchangeably in this dissertation. Other actors can send a message to this `ActorRef`, and the actor identified

by this `ActorRef` will receive this message. Note that other actors *cannot* directly access the local state of this actor (e.g., access fields, call instance methods) through the `ActorRef`.

Acquaintance Relations. Actor `A` knows of actor `B` if `A` has access to the actor reference (`ActorRef`) of `B`. At Line 5 in our example, we create a `clientActor` and pass it the `ActorRef` of the `serverActor` (now the client knows of the server and can send messages to it). The actor reference can also be sent as a message to inform other actors. Another type of acquaintance between actors is via receiving messages: when an actor receives a message, it can access the actor reference of the sender through the `getSender()` method. An actor can also get its own actor reference through the `getSelf()` method.

Sending and Processing Messages. Every actor must implement a message handler, the `onReceive` method. The `onReceive` method takes as input a message object, and is invoked upon receiving a message. Typically, different types of messages trigger different behaviors in the actor. For example, the `onReceive` of the `Client` actor (Lines 13-23) behaves differently on the `WithdrawMessage` and the `DepositMessage`. Messages are sent via calling the `tell` method on an `ActorRef` object (e.g., Line 17).

3.1.3 Problem Description

Actors model an *open system* – a system that may interact with its external environment. In order to preserve locality properties of actors, such interaction is through messages received by receptionist actors in the system and messages sent to external actors by actors in the system. Thus the entry points of the system are message handlers of receptionists. Examples of open systems in the real-world include Twitter, LinkedIn, Facebook Chat, and Halo 4, all of which have been implemented using actors.

The input to our problem includes: (1) the code under test, (2) a target code location, (3) a user defined set of receptionists of the system, and (4) a start configuration defining the initial acquaintance between actors. The output (if found) is a test case that covers the target. Such a test consists of messages sent to relevant actors as well as their arrival orders on each of these actors.

In our Bank Account example, the code under test is the `Client` and the `Server` actor classes; the receptionist is the `Client` actor as the client is the interface of the system for user interactions. The main method sets up the initial acquaintance that the client knows of the server. Suppose our target is the negation of the assertion at Line 32. One possible output test case that covers the target is as follows. The client receives a deposit message with the amount 50 and a withdraw message with the amount 120, in that order. Since the deposit message is received before the withdraw message, the condition at Line 15 is

evaluated to true, and the client forwards both messages to the server. However, on the server side, the withdraw message somehow arrives before the deposit message, causing the assertion violation. This test case specifies the messages received by the client and the server as well as the message receiving orders on both actors. For illustration purposes, we do not assume the first-in-first-out (FIFO) message delivery between a pair of actors in this example. Given FIFO message delivery, the two messages could be routed through different actors, still creating nondeterminism in the arrival order at the server.

Note that we must specify the receptionists of an actor system in our problem settings. This requirement enforces system-level testing because internal actors can only be tested through receptionists. In our example, to cover the target we have to send messages to the client in order to trigger messages sent to the server. If all actors were potential receptionists, then every actor may receive messages directly from the external environment. In this case, each actor may be tested individually with all possible message sequences and no interaction between actors need be considered. The benefit of considering external messages only to designated actors is that it constrains the generated tests to those which would realistically occur in an actor system. While this means that system-level testing is required, it eliminates consideration of tests based on arbitrary messages to individual actors that would never be sent in a realistic system.

3.2 ACTOR LANGUAGE

To formally describe our method, we define a simplified actor language by extending Featherweight Java [60] and adding actor constructs to it. We choose the Featherweight Java language for its simplicity and for the fact that our tool targets Java *Akka*. The formalism in this chapter largely follows the conventions in previous work [60, 61, 62]. The actor constructs in our language resemble the counterparts in Java *Akka*. Although there have been formalizations of actor languages [2, 63], our formalization of the language is closely coupled with our analysis, and includes more details such as data store and context, which are required to specify our analysis.

3.2.1 Syntax

Figure 3.2 describes the grammar of a simplified actor language. The language is in A-Normal form, where computations are syntactically sequentialized. For example, the statement `v = o.m(o.f)` is transformed to two statements `v1 = o.f; v2 = o.m(v1)` in A-Normal form. Such transformation brings our language closer to an intermediate language

$$
\begin{aligned}
\textsf{Class} &::= \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{C''\ f;}\ K\ \overrightarrow{M}\} \\
\textbf{ActorClass} &\textbf{::=} \texttt{class}\ \textbf{C}\ \texttt{extends}\ \textbf{C'}\ \{\overrightarrow{\textbf{C''}\ \textbf{f;}}\ \textbf{K}\ \textbf{R}\ \overrightarrow{\textbf{M}}\textbf{\}} \\
K \in \textsf{Ctor} &::= C(\overrightarrow{C'\ f})\ \{\texttt{super}(\overrightarrow{f'});\ \overrightarrow{\texttt{this}.f'' = f'''};\} \\
\textbf{R} \in \textbf{Receive} &\textbf{::=} \texttt{void}\ \texttt{onReceive}\textbf{(C v) \{}\overrightarrow{\textbf{C' v';}}\ \overrightarrow{\textbf{s}}\textbf{\}} \\
M \in \textsf{Method} &::= C\ m(\overrightarrow{C'\ v})\ \{\overrightarrow{C'v';}\ \overrightarrow{s}\} \\
s \in \textsf{Stmt} &::= v = e;^{\ell} \mid \texttt{return}\ v;^{\ell} \mid \texttt{if}\ (e)\ \overrightarrow{s}\ \texttt{else}\ \overrightarrow{s'};^{\ell} \mid \textbf{v.send(v');}^{\ell} \\
e \in \textsf{Expr} &::= v \mid (C)\ v' \mid v.f \mid v.m(\overrightarrow{v'}) \mid \texttt{new}\ C(\overrightarrow{v}) \mid v\ \textsf{op}\ v' \mid \textbf{aref} \\
\textbf{aref} \in \textbf{ARef} &\textbf{::=} \texttt{create(C.class,}\ \overrightarrow{\textbf{v}}\texttt{)} \text{ — } \texttt{self} \text{ — } \texttt{sender} \\
v &\in \textsf{Var} \text{ is a set of variable names} \\
f &\in \textsf{FieldName} \text{ is a set of field names} \\
C &\in \textsf{ClassName} \text{ is a set of class names} \\
m &\in \textsf{MethName} \text{ is a set of method names} \\
\ell &\in \textsf{Lab} \text{ is a set of labels} \\
\textsf{op} &\in \{+, -, *, /, <, >, ==, !=, \ldots, \texttt{instanceof}\}
\end{aligned}
$$

Figure 3.2: An actor language extending Featherweight Java.

for simpler semantics definitions. Most of the notations in Featherweight Java are intuitive. We give a quick reminder of the less obvious conventions. A class declaration consists of a list of fields (we use an arrow to represent a list), a single constructor, and a list of methods. The constructor takes as input a list of arguments and assigns each argument to the corresponding field. Each statement in the language is assigned a distinct label. We augment Featherweight Java with *binary* expressions and *if* statements, which are later needed in the formalization of the BSE semantics. We omit the *loop* statement because loops are bounded and unrolled into *if* statements in our analysis. Such unrolling trades completeness for tractability and is standard practice in testing.

We now introduce actor constructs (highlighted in bold). Each actor class declaration must include exactly one `onReceive` method. This method takes a single input (message) and returns `void`. An actor creation operation $\texttt{create}(A.\texttt{class},\ \overrightarrow{v})$ takes as input the class of the actor to be created C, followed by a list of arguments to the constructor of C, and returns the actor reference of the created actor. A message send operation $v.\texttt{send}(v');^{\ell}$ sends the message v' to the actor reference v of the recipient actor.

$$
\begin{aligned}
\alpha \in ActorMap &= ActorRef \to ActorState \\
msg \in Message &= ActorRef \times ActorRef \times Obj \\
r \in ActorRef &\subset Obj \\
\varsigma \in ActorState &= \mathsf{Stmt} \times Stack \times Store \times CallStack \times Context \\
st \in Stack &= (\mathsf{Var} \rightharpoonup Addr)^* \\
\sigma \in Store &= Addr \to Obj \\
o \in Obj &= HContext \times (\mathsf{FieldName} \rightharpoonup Addr) \\
cs \in CallStack &= (\mathsf{Stmt} \times Context \times Addr)^* \\
a \in Addr &= (\mathsf{Var} \times Context) \cup (\mathsf{FieldName} \times HContext) \\
c \in Context &\text{ is an infinite set of regular contexts} \\
hc \in HContext &\text{ is an infinite set of heap contexts}
\end{aligned}
$$

Figure 3.3: Domains of actor maps and messages.

3.2.2 Concrete Semantics

An instantaneous snapshot of an actor systems is called a *configuration*.[2] The semantics of our language is defined by a transition relation on configurations. A configuration is a tuple,

$$\langle \alpha \mid \mu \rangle \tag{3.1}$$

where α is an actor map that maps a finite set of actor references to actor states, and μ is a finite multi-set of pending messages. It is important to note that by modeling the pending messages as a multi-set, the order in which messages are sent is not preserved. As a result, our language semantics does not guarantee the FIFO message delivery between a pair of actors. We choose not to assume the FIFO message delivery in both the concrete language semantics and the BSE semantics in Section 3.4.2, because the FIFO semantics is not primitive in the Actor model [59, 1, 2]. However, one can easily accommodate the FIFO semantics in our models by replacing the multi-set with a data structure that preserves the message sending orders (e.g., a set of lists representing a sequence of messages, one list for each pair of a sender and a receiver). Since most real-world actor languages and frameworks guarantee the FIFO message delivery, we do implement the FIFO semantics in our tool.

The domains in a configuration are described in Figure 3.3. A message is a tuple consisting of the actor reference of the sender, the actor reference of the recipient, and the message

[2]Recall that actors are asynchronous: there is no unique global time. Thus an actor snapshot is with respect to some frame of reference, i.e., a causally consistent linearlization of a partial order.

Actor Creation

$$\langle \alpha \bullet r \mapsto (\, [\![v = \texttt{create}(C\texttt{.class}, \overrightarrow{v'})\texttt{;}^{\ell}]\!], st, \sigma, \_\,) \mid \_ \rangle \Rightarrow_C \langle \alpha \bullet r \mapsto (\, succ(\ell), st, \sigma', \_\,) \bullet r' \mapsto \varsigma \mid \_ \rangle, \text{ where}$$

r' is fresh $\quad \sigma' = \sigma + [st(v) \mapsto r'] \quad o'_i = \sigma(st(v'_i)) \quad \varsigma = (\texttt{nil}, [], \sigma'', [], \texttt{nil})$

$\overrightarrow{f} = \mathcal{F}(C) \quad a_i = (f_i, hc) \quad o = (hc, [f_i \mapsto a_i]) \quad \sigma'' = [a_{this} \mapsto o, a_i \mapsto o'_i, a_{self} \mapsto r']$

Message Sending

$$\langle \alpha \bullet r \mapsto (\, [\![v\texttt{.send}(v')\texttt{;}^{\ell}]\!], st, \sigma, \_\,) \mid \mu \rangle \Rightarrow_C \langle \alpha \bullet r \mapsto (\, succ(\ell), st, \sigma, \_\,) \mid \mu \bullet \big(r, \sigma(st(v)), \sigma(st(v'))\big) \rangle$$

Message Receiving

$$\langle \alpha \bullet r \mapsto (\, \texttt{nil}, st, \sigma, cs, c\,) \mid \mu \bullet (r', r, o) \rangle \Rightarrow_C \langle \alpha \bullet r \mapsto (\, s, st', \sigma', cs', c'\,) \mid \mu \rangle, \text{ where}$$

c' is fresh $\quad o_0 = \sigma(a_{this}) \quad [\![\texttt{void onReceive } (C\ v)\ \{\overrightarrow{C'\ v'};\ \overrightarrow{s'}\}]\!] = rec(cls(o_0))$

$s = car(\overrightarrow{s'}) \quad a = (v, c') \quad a'_i = (v'_i, c') \quad st' = cons([v \mapsto a, v'_i \mapsto a'_i], st)$

$cs' = cons((\texttt{nil}, c, \texttt{nil}),\ cs) \quad \sigma' = \sigma + [a \mapsto o,\ a_{sender} \mapsto r']$

Self Reference

$$\langle \alpha \bullet r \mapsto (\, [\![v = \texttt{self}\texttt{;}^{\ell}]\!], st, \sigma, \_\,) \mid \_ \rangle \Rightarrow_C \langle \alpha \bullet r \mapsto (\, succ(\ell), st, \sigma + [st(v) \mapsto r)], \_\,) \mid \_ \rangle$$

Sender Reference

$$\langle \alpha \bullet r \mapsto (\, [\![v = \texttt{sender}\texttt{;}^{\ell}]\!], st, \sigma, \_\,) \mid \_ \rangle \Rightarrow_C \langle \alpha \bullet r \mapsto (\, succ(\ell), st, \sigma + [st(v) \mapsto \sigma(a_{sender})], \_\,) \mid \_ \rangle$$

Figure 3.4: Concrete semantics for actor operations of the simplified actor language.

content. An actor reference is an object that stores the location information of an actor. An actor state ς consists of a statement under execution, a data stack to store local variables, a data store of points-to relations, a call stack to track active method invocations, and a current execution context. A data stack st consists of a list of data frames, each of which maps local variables to addresses. A data store σ maps addresses to objects. A call stack cs consists of a list of call frames, and each call frame consists of the statement to return to, the context to restore, and the address to store the return value. An object o consists of a heap context and a list of fields. An address is a location that holds an object. An address a consists of either a local variable and its regular context (allocated for a local variable) or a field and its heap context (allocated for a field). In the concrete semantics, every dynamic

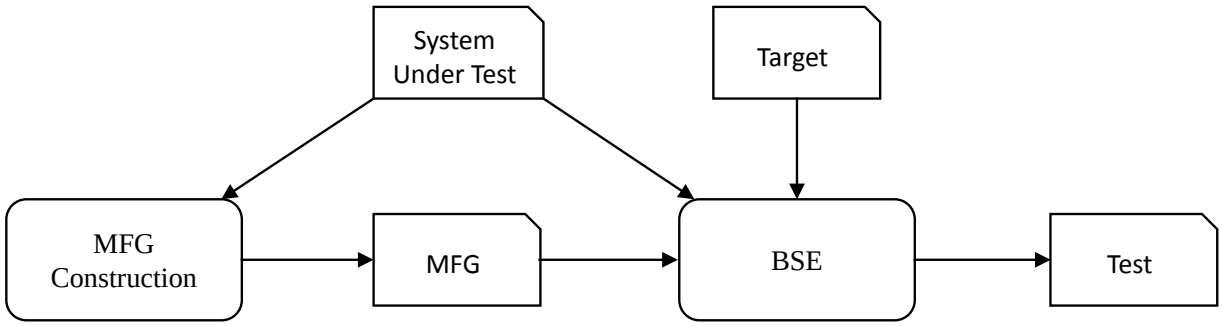


Figure 3.5: The overview of our two-phased test generation method.

object instance has a unique heap context, and every dynamic method call has a unique regular context.

We express the concrete semantics of our language as a transition relation ($\Rightarrow_C$) from one configuration to another. Figure 3.4 shows the semantics of actor operations only. The semantics of local computations in an actor is similar to the normal semantics of Java, and thus omitted. For simplicity, we use underscore _ in our transition rules, to represent the remaining states in a tuple that are neither used nor updated in the transition. We use standard functions $car, cdr, cons, list$ to manipulate lists, and define a number of helper functions: *succ* returns the next statement given the label of the current statement, $\mathcal{F}$ returns a list of field names for a given class, *cls* returns the class name of a given object, and *rec* returns the declaration of the `onReceive` method of a given class. We use the operator $\bullet$ to add an element to a set, and the notation $+$ to insert or update (if existing) entries in a map. We use `nil` as the null value for every domain. A *fresh* value means that a new value is generated from the corresponding domain. The symbols a_{this}, a_{self}, and a_{sender} represent reserved addresses to store the *this* object, the actor reference of itself, and the actor reference of a sender, respectively.

The *Actor Creation* rule says that a new actor is created with a fresh reference r' in the system. The actor has an initial state, where the current statement is `nil`. The *Message Sending* rule defines the asynchronous semantics of sending messages. The new message is put in the set of pending messages μ, and the sending actor continues its execution. Note that messages are immutable so that there are no concurrent writes on messages. The *Message Receiving* rule says that an actor can receive a message only when it is ready (i.e., the statement is `nil`). Upon receiving the message, the `onReceive` method is invoked, and the message is no longer pending and thus removed from μ. After executing the `onReceive` method, the statement is set to `nil`, signifying that the actor is ready to receive a message again. The *Self Reference* rule says that the actor reference of the *this* object is assigned to

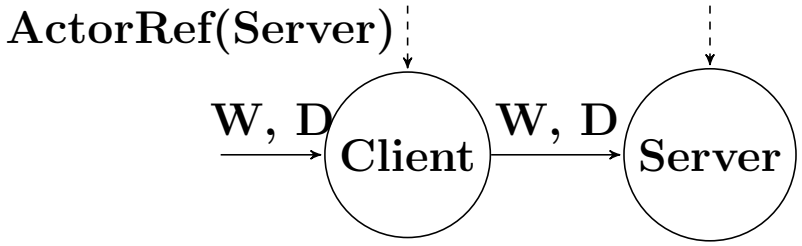


Figure 3.6: The MFG of the Bank Account example. The symbols W and D represent the withdraw message and the deposit message, respectively.

a local variable v. Similarly, the *Sender Reference* rule says that the actor reference of the message sender is assigned to a local variable v.

3.3 MESSAGE FLOW GRAPH CONSTRUCTION

Our test generation method operates in two phases as shown in Figure 3.5. In the first phase, we use static analysis to construct a *message flow graph* (MFG), an abstraction of an actor system that models potential interactions (i.e., actor creation and communication) between actors in the system. The input to our MFG analysis is the system under test including the code and the specified receptionists, and the output is an MFG of the system. In the second phase, we use BSE to generate a test that covers a given target. To generate tests that exercise multiple actors, BSE must go across actors. The MFG from the first phase is a key input that enables *inter-actor* BSE. After BSE reaches the entry of the message handler on an actor a, it queries the MFG to obtain actors that can send the required message to the actor a. Then BSE picks one potential sender, jumps to the exit of the message handler of the sender, and continues with the previous path constraint carried over. When a feasible path is found during the path exploration, we generate the test from the path constraint. We explain the MFG construction and the BSE (in Section 3.4) in details.

An MFG is a directed graph between *abstract* objects, where an abstract object represents multiple concrete objects of the same class whose field values have been merged into a set. Specifically, a node in the MFG represents an abstract actor and a directed edge between two nodes means that the abstract actor represented by the source node either creates or sends a message to the abstract actor represented by the sink node. MFG edges are labeled with abstract constructor parameters for actor-creation edges and abstract messages for message-sending edges. Note that the MFG edges do not indicate the acquaintance between actors–it is possible that an actor a knows of another actor b, but there is no edge from a to b because

$$
\begin{aligned}
\omega \in \Omega &= ActorState \times Graph \times RefMap \\
\gamma \in RefMap &= ActorRef \to (\mathsf{ClassName} \times Obj) \\
r \in ActorRef &\subset Obj \\
G \in Graph &= ActorRef \times ActorRef \times Type \rightharpoonup (\mathcal{P}(Obj))^* \\
Type &= \{\texttt{create}, \texttt{send}\} \\
\varsigma \in ActorState &= \mathsf{Stmt} \times Stack \times Store \times CallStack \times Context \\
st \in Stack &= (\mathsf{Var} \rightharpoonup Addr)^* \\
\sigma \in Store &= Addr \to \mathcal{P}(Obj) \\
o \in Obj &= HContext \times (\mathsf{FieldName} \rightharpoonup Addr) \\
cs \in CallStack &= (\mathsf{Stmt} \times Context \times Addr)^* \\
a \in addr &= (\mathsf{Var} \times \mathsf{MethodName} \times Context) \cup (\mathsf{FieldName} \times HContext) \\
Context &= HContext = \mathsf{Lab}
\end{aligned}
$$

Figure 3.7: State space of the small-step state machine.

a neither creates b nor sends a message to b.

An abstract object may be replaced by its class if there is only one abstract object per class. Figure 3.6 shows the MFG of the Bank Account example. There are two actors, `Client` and `Server`, in the graph. The symbols W and D represent the `WithdrawMessage` and the `DepositMessage`, respectively. Both actors are created (creation edges are represented with dashed arrows) by the external environment. The `Client` is initialized with an actor reference mapped to the `Server`, and it can send `WithdrawMessage` and `DepositMessage` to the `Server`. The `Client` is the only receptionist of the system and can receive `WithdrawMessage` and `DepositMessage` from the external environment.

To construct a MFG, we need to not only resolve the recipient of each message-sending site and the actor being created of each actor-creation site, but also pass along the messages and constructor parameters between actors. This is because the message and constructor parameters can affect the analysis of receiving actors. We use points-to analysis to compute the points-to sets for messages, constructor parameters, and actor references. In addition, we model the semantics of actor operations so that analysis information can be carried across actor boundaries. In particular, the actor creation operation conceptually creates two objects: an actor reference object and a corresponding actor object. Our analysis keeps track of such mappings to resolve the actor being created, and passes the points-to set of constructor parameters to this actor for instantiation.

Note that passing only the type of the message or constructor parameter between actors

can result in unacceptable imprecision in our analysis. For example, a common case is that an actor reference `r` is sent as a message to a recipient actor `A`; `A` receives `r` and then sends a message to `r`. When resolving `r` in `A`, we only know that the type of `r` is `ActorRef`, but we know nothing about the actor that lives in `r`. Thus, we have to conservatively assume all actor classes in our system may live in `r`, and add a message-sending edge from `A` to every actor class. To avoid such imprecision, we need to pass along the points-to sets of messages and constructor parameters instead of their types. We next formally describe our analysis.

3.3.1 Analysis Semantics

We express the semantics of our analysis using small-step state machines, each modeling one abstract actor. Communication between actors is modeled by global states shared across state machines. The domain Ω of a state machine is defined in Figure 3.7. The reference map γ stores the mappings between actor references and the actors created in the system. The graph G records the actor-creation and message-sending events between actors. Specifically, G maps a tuple of a source actor reference, a sink actor reference, and an operation type to a list of points-to sets of messages or constructor parameters. Visually, an entry in the map can be seen as a directed edge, with the label being the list of points-to sets. The *ActorState* is similar to the one defined in concrete semantics of our language except that now the *ActorState* is an abstract state: the store maps an address to a set of objects rather than one object; the regular and heap contexts are a finite set of statement labels. An important design decision made by our analysis is that we create only one abstract actor object per actor class. That is, actors of the same class created in different sites are merged into one abstract actor object by merging the points-to sets of the corresponding fields. In this way, we only need to create one state machine per actor class, making our analysis faster and more scalable. The incurred imprecision can be refined by the BSE in phase II because our BSE distinguishes every concrete actor.

The analysis semantics is defined by the transition relation $(\Rightarrow_A) \subset \Omega \times \Omega$. The analysis semantics of local computations is precisely the 1-object-sensitive points-to analysis [64]. Figure 3.9 shows the transition rules for local computations in the MFG analysis. Since local computations concern only the actor state ς in ω, we omit other states in ω in our transition rules for better readability (the other states are the same on both sides of the rules). The operator $\sqcup$ is used to merge two maps by merging the values of the same key in both maps. The *dispatch* function takes as input an object and a method name, and returns the dispatched method[3]. The transition rules describe precisely the 1-object-sensitive points-

[3]Our language does not support method overloading, and thus a method can be dispatched based on the given object and its method name

Actor Creation

$\big((\llbracket v = \texttt{create}(C\texttt{.class}, \overrightarrow{v'})\texttt{;}^{\ell} \rrbracket, st, \sigma, \_), G, \gamma\big) \Rightarrow_A \big((succ(\ell), st, \sigma', \_), G', \gamma'\big)$, where

$(\gamma', r) = getRef(\gamma, C) \qquad \sigma' = \sigma \sqcup [st(v) \mapsto \{r\}] \qquad r' \in \sigma(a_{self})$

$G' = merge(\, G, [(r', r, \texttt{create}) \mapsto list(\sigma(st(v'_i)))] \,)$

Message Sending

$\big((\llbracket v\texttt{.send}(v')\texttt{;}^{\ell} \rrbracket, st, \sigma, \_), G, \_\big) \Rightarrow_A \big((succ(\ell), st, \sigma, \_), G', \_\big)$, where

$r \in \sigma(st(v)) \qquad r' \in \sigma(a_{self}) \qquad G' = merge(\, G, \, [(r', r, \texttt{send}) \mapsto list(\sigma(st(v')))] \,)$

Message Receiving

$\big((\texttt{nil}, st, \sigma, cs, c), G, \_\big) \Rightarrow_A \big((s, st', \sigma', cs', c'), G, \_\big)$, where

$o_0 \in \sigma(a_{this}) \qquad \llbracket \texttt{void onReceive}\ (C\ v)\ \{\overrightarrow{C'\ v'};\ \overrightarrow{s}\} \rrbracket = rec(cls(o_0)) \qquad s = car(\overrightarrow{s})$

$(hc_0, \_) = o_0 \qquad c' = hc_0 \qquad a = (v, \texttt{onReceive}, c') \qquad a'_i = (v'_i, \texttt{onReceive}, c')$

$st' = cons([v \mapsto a, v'_i \mapsto a'_i], st) \quad cs' = cons((\texttt{nop}, c, \texttt{nil}), cs) \qquad r \in \sigma(a_{self})$

$O_r = preds(G, r, \texttt{send}, \gamma) \qquad O \in \{car(G((r', r, \texttt{send}))) \mid r' \in O_r\}$

$\sigma' = \sigma \sqcup [a \mapsto O, a_{sender} \mapsto O_r]$

Self Reference

$\big((\llbracket v = \texttt{self}\texttt{;}^{\ell} \rrbracket, st, \sigma, \_), \_\big) \Rightarrow_A \big((succ(\ell), st, \sigma \sqcup [st(v) \mapsto \sigma(a_{self})], \_), \_\big)$

Sender Reference

$\big((\llbracket v = \texttt{sender}\texttt{;}^{\ell} \rrbracket, st, \sigma, \_), \_\big) \Rightarrow_A \big((succ(\ell), st, \sigma \sqcup [st(v) \mapsto \sigma(a_{sender})], \_), \_\big)$

Figure 3.8: Abstract semantics for actor operations in MFG analysis

to analysis [64]. The *Object Allocation* rule says that the heap context of an object is the label of its allocation site. The *Method Invocation* rule describes the context sensitivity. The rule says that the context used for analyzing a method is the heap context of the receiver object, which is the label of its allocation site.

Figure 3.8 describes transition rules for the actor operations. The $getRef$ function checks if the given class C is in the value set of γ. If found, it returns itself and the key of the value. If not found, it adds an entry $r \mapsto (C, \texttt{nil})$ to γ, where r is fresh, and returns the updated γ' and r. Since only one abstract actor object is created per actor class, an actor class can appear in at most one tuple in the value set of γ. The $merge$ function merges the labels of edges with the same source and sink. The $preds$ function finds all predecessors of a given type for a node r in the graph G and returns the set of actor objects mapped by the predecessors in γ.

In the *Actor Creation* rule, instead of instantiating the actor object at the creation site,

Variable Reference

$([\![v = v'\,;^{\ell}]\!], st, \sigma, \_) \Rightarrow_A (succ(\ell), st, \sigma', \_),$ where $\quad \sigma' = \sigma \sqcup [st(v) \mapsto \sigma(st(v'))]$

Field Reference

$([\![v = v'.f\,;^{\ell}]\!], st, \sigma, \_) \Rightarrow_A (succ(\ell), st, \sigma', \_),$ where

$(\_, [f \mapsto a_f]) \in \sigma(st(v')) \qquad \sigma' = \sigma \sqcup [st(v) \mapsto \sigma(a_f)]$

Object Allocation

$([\![v = \texttt{new}\ C(\overrightarrow{v'})\,;^{\ell}]\!], st, \sigma, \_) \Rightarrow_A (succ(\ell), st, \sigma', \_),$ where

$hc = \ell \quad \overrightarrow{f} = \mathcal{F}(C) \quad a_i = (f_i, hc) \quad o = (hc, [f_i \mapsto a_i])$

$\sigma' = \sigma \sqcup [st(v) \mapsto \{o\}, a_i \mapsto \sigma(st(v_i'))]$

Method Invocation

$([\![v = v_0.m(\overrightarrow{v'})\,;^{\ell}]\!], st, \sigma, cs, c) \Rightarrow_A (s, st', \sigma', cs', c'),$ where

$M = [\![C\ m(\overrightarrow{C'\ v''})\ \{\overrightarrow{C''v'''};\ \overrightarrow{s'}\}]\!] = dispatch(o_0, m) \qquad o_0 \in \sigma(st(v_0)) \qquad (hc_0, \_) = o_0$

$c' = hc_0 \qquad a_i = (v_i'', m, c') \qquad a_i' = (v_i''', m, c') \qquad st' = cons([v_i'' \mapsto a_i, v_i''' \mapsto a_i'],\ st)$

$s = car(\overrightarrow{s'}) \qquad \sigma' = \sigma \sqcup [a_i \mapsto \sigma(st(v_i'))] \qquad cs' = cons((succ(\ell), c, st(v)),\ cs)$

Return

$([\![\texttt{return}\ v\,;^{\ell}]\!], st, \sigma, cs, c) \Rightarrow_A (s, cdr(st), \sigma', cdr(cs), c'),$ where

$(s, c', a_{ret}) = car(cs) \qquad \sigma' = \sigma \sqcup [a_{ret} \mapsto \sigma(st(v))]$

Casting

$([\![v = (C)\ v'\,;^{\ell}]\!], st, \sigma, \_) \Rightarrow_A (succ(\ell), st, \sigma', \_),$ where $\quad \sigma' = \sigma \sqcup [st(v) \mapsto \sigma(st(v'))]$

Figure 3.9: Abstract semantics for local computations in MFG analysis.

an actor-creation event is recorded and merged into the graph. Subsequently, when a state machine for this actor class is created, actor-creation events are used to instantiate the single abstract actor object for this class. Similarly in the *Message Sending* rule, a message-sending event is recorded and merged into the graph. The *Message Receiving* rule says that the *onReceive* method of the actor is invoked upon receiving a message. The graph G is queried to find the set of all possible senders O_r, and the set of all possible messages received by O. Note that when updating the call stack, we use `nop` instead of `nil` for the statement to return to. `nop` indicates no operation to be performed and stops the state machine. Otherwise, the state machine will not halt.

3.3.2 MFG Construction Algorithm

Algorithm 3.1 shows our iterative algorithm to construct the MFG. The algorithm takes as input an actor system P, a raw graph G, and a reference map γ, and outputs an MFG graph. G and γ are initialized from the driver code that sets up the actor system. Initially, G contains actor-creation and message-sending events by the external environment, and γ contains the mappings for actors created by the external environment. For each actor class, one state machine is instantiated to model the abstract actor of this class. The algorithm maintains a *worklist* that keeps track of the abstract actors to be analyzed next as well as a $factStore$ that stores the relevant data facts for each abstract actor. The data facts for an abstract actor are essentially the set of incoming edges of this actor node in G, and these facts affect the initial state of the state machine for this actor.

Algorithm 3.1: Iterative MFG construction

Input : An Actor system P, a raw graph $G \in Graph$, and an actor reference map $\gamma \in RefMap$

Output: A message flow graph of P

```
1  worklist ← [ ]      factStore ← [ ]
2  worklist.appendAll( γ.keySet() )
3  while worklist not empty do
4      r ← worklist.removeFirst()
5      beforeFacts ← InEdges (r, G)
6      if factStore[r] ≠ beforeFacts then
7          factStore[r] ← beforeFacts
8          M_r ← CreateStateMachine (r, G, γ)
9          M_r.execute()
10         worklist.appendAll( Successors (r, G) )
11 return CollapseToMFG (γ, G)
12 Procedure CreateStateMachine (r, G, γ)
13     (C, _) ← γ(r)      f⃗ ← F(C)      a_i ← (f_i, ℓ_C)
14     o ← (ℓ_C, [a_i ↦ f_i])                          // actor allocation
15     γ ← γ + [r ↦ (C, o)]                            // ref map update
16     O⃗ ← [∅, ..., ∅]                                // a list of points-to sets
17     foreach (r', r'', create) ↦ O⃗' in InEdges (r, G) do
18         O_i ← O_i ∪ O'_i
19     σ ← [a_this ↦ {o}, a_i ↦ O_i, a_self ↦ {r}]
20     ω_0 ← ((nil, [ ], σ, [ ], nil), G, γ)
21     Create M_r with the initial state ω_0
22     return M_r
```

The algorithm starts with pushing the initial actors onto the *worklist* (Line 2), and iteratively analyzes these actors one at a time. Before the analysis, the algorithm computes

the relevant data facts for this actor from G (Line 5). It then checks whether the facts are changed, by comparing the computed facts with the previous facts stored in $factStore$. If changed, the algorithm updates the facts for this actor in $factStore$ (Line 7), analyzes this actor with these new facts by instantiating and running the state machine described in Section 3.3.1 (Lines 8-9), and pushes all the successors of this actor node onto *worklist* (Line 10). Otherwise, the algorithm skips this actor because the execution of its state machine will yield the same result and will not change the global state G. This process continues until *worklist* is empty, indicating a fixed point is reached. The `CreateStateMachine` procedure is the only place where instantiations of abstract actors happen. The constructor parameters of multiple actor-creation edges are merged (Lines 18-21) and the results are used to initialize the fields of the abstract object (Line 22). Finally, the algorithm builds an MFG from G and γ by collapsing the abstract objects of nodes and labels into classes. If an object is an actor reference, we also encode the class of the underlying actor into the MFG.

3.3.3 Optimizations

Our analysis applies two lightweight yet effective optimizations to actor classes based on the code pattern in actor programs. Since actors often receive multiple types of messages and behave differently for each message type, a common code pattern in actors' `onReceive` methods is that an *if* statement is used at the top of its control flow to check the message type and process one type of message in one branch. In our running example, both the `Client` and the `Server` actors follow this pattern.

Our first optimization eliminates unreachable code based on the potential types of the message in our analysis. Specifically, we compute the potential types from the points-to set of the message and analyze only the branches of the top *if* statement that may be taken under these message types. Our second optimization is based on the idea that when a message must be of a certain type under some context, we can safely remove objects that are not an instance of this type from the points-to set of this message. The optimization works as follows: after entering a branch of the top *if* statement, we carry the corresponding type constraint of the message (obtained from the condition of the *if* statement) with our analysis. That is, whenever we query the points-to set of the message in this branch, an additional filter function $f : \mathcal{P}(Obj) \times \mathsf{ClassName} \rightarrow \mathcal{P}(Obj)$ is applied to the original points-to set to filter out objects that are not an instance of the given type. Our evaluation shows that these optimizations significantly reduce the size of the MFGs.

Example. Let us illustrate the optimizations using the `Client` actor in Figure 3.1. Suppose that the points-to set of the `message` parameter in the `onReceive` method contains only

$$
\begin{aligned}
\alpha \in ActorMap &= ActorRef \rightarrow ActorState \\
Event &= SendingEvent \cup CreationEvent \\
SendingEvent &= \widehat{ActorRef} \times \widehat{ActorRef} \times \widehat{Var} \times \widehat{Time} \\
CreationEvent &= \widehat{ActorRef} \times \mathsf{ClassName} \times (\widehat{Var})^* \times \widehat{Time} \\
\varsigma \in ActorState &= LocalState \times \widehat{Time} \times Requests \\
\beta \in LocalState &= \mathsf{Stmt} \times CallStack \times \widehat{Var} \\
cs \in CallStack &= (\mathsf{Stmt} \times \widehat{Var} \times \widehat{Var})^* \\
Q \in Requests &= \widehat{Var} \times \widehat{ActorRef} \times \widehat{Time}
\end{aligned}
$$

$\widehat{Var}, \widehat{ActorRef}, \widehat{Time}$ are sets of free variables in first order logic.

Figure 3.10: State space of the backward symbolic execution.

one `DepositMessage` message. Based on the first optimization, we only need to analyze the second branch of the *if* statement (Lines 20 - 22) instead of the whole method. To illustrate our second optimization, we now suppose that the points-to set of the `message` parameter contains a `WithdrawMessage` message and a `DepositMessage` message. Then both branches of the *if* statement must be analyzed. When analyzing its first branch (Lines 14-18), we know that the `message` parameter must be of the type `WithdrawMessage`. With this type constraint, we can remove the `DepositMessage` message from the points-to set in this branch because it is not an instance of the type `WithdrawMessage`. Hence, we can conclude that at Line 17, `message` must point to a `WithdrawMessage` message rather than may point to a `WithdrawMessage` message or a `DepositMessage` message. Similarly, the optimization can be applied to the second branch as well.

3.4 BACKWARD SYMBOLIC EXECUTION

In phase II, we use backward symbolic execution to generate tests for the target. BSE starts from the target, and performs a backward exploration, searching for a feasible path to the entry points of the system. Constraints over the execution are collected and used to generate the test. The generated test consists of the messages sent to relevant actors as well as the message receiving orders.

The semantics of BSE is formally defined as a transition relation $\Rightarrow_S$ from one symbolic

configuration to another symbolic configuration. A symbolic configuration is a tuple,

$$\langle \alpha \mid \mu \mid \phi \mid \chi \rangle \tag{3.2}$$

where α represents relevant actors in BSE and is a map from a finite set of actor references to actor states, μ is a finite set of pending events (including both actor creation and message sending events). ϕ is the path condition collected over the transitions, and χ is the set of external messages to the system. The domain of ϕ is the quantifier-free formulae in *first-order logic* (FOL) with equality. The domain of the remaining configuration is described in Figure 3.10. Note that $\widehat{Var}, \widehat{ActorRef}, \widehat{Time}$ are sets of free variables in FOL, which can hold values of primitives and references. A message-sending event consists of the actor reference of the sender, the actor reference of the recipient, the message, and the time when the message is sent. An actor-creation event consists of the actor reference of the actor being created, the type of the actor, and a list of constructor parameters, and the creation time. An actor state consists of a local state, the current local time of the actor, and a set of message requests.

Since BSE goes backwards, a message request under this context indicates that a certain message is required in order for the execution to reach this point, yet this message is not in the mailbox of that actor. For each message request, BSE attempts to find an actor that can send the corresponding message, and thus "fulfill" this request. The local state consists of the current statement, the call stack, and a variable representing the receiver object of the current method call. A message request consists of a message, an actor reference for the sender, and the time of receiving the message. The call stack consists of a list of call frames, and each call frame consists of the statement to return to, the variable of the return value, and the variable of the caller object. $\widehat{Time}$ is a set of integer variables.

To describe the BSE semantics, we add two additional types of statements to our language as indicators of reaching the entry of a method. We use $\texttt{entry}_R\texttt{;}$ as the first statement for every `onReceive` method, and use `entry;` as the first statement for all other methods. We next formally define the semantics of local computations for intra-actor BSE as well as the semantics of actor operations for intra-actor BSE.

3.4.1 Semantic of Local Computations in BSE

Figure 3.11 and Figure 3.12 show respectively the semantics of intra-procedural BSE and the semantics of inter-procedural BSE for local computations in an actor. Since local computations concern only the local state β and the path condition ϕ, we omit other states

Variable Reference

$((\llbracket v = v' \,;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi[\hat{v'}/\hat{v}])$

Binary Expression

$((\llbracket v = v' \texttt{ op } v'' \,;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi'),$ where $\phi' = \phi[(\hat{v'} \texttt{ op } \hat{v''})/\hat{v}]$

Field Reference

$((\llbracket v = v'.f \,;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi[read(\hat{v'}, f)/\hat{v}])$

Field Update

$((\llbracket v.f = v' \,;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi[update(f, v, v')/f])$

Casting

$((\llbracket v = (C)\ v' \,;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi[\hat{v'}/\hat{v}] \wedge subType(type(\hat{v}), C))$

Object Allocation

$((\llbracket v = \texttt{new } C(\overrightarrow{v'}) \,;^{\ell} \rrbracket, \_), \phi) \Rightarrow_S ((pred(\ell), \_), \phi'),$ where

$\hat{v''}$ is fresh $\quad \overrightarrow{f} = \mathcal{F}(C) \quad \phi' = \phi[\hat{v''}/\hat{v},\ update(f_i, v'', v_i')/f_i] \wedge type(\hat{v''}) == C$

If-True

$((\llbracket \texttt{if } (e)\ \overrightarrow{s} \texttt{ else } \overrightarrow{s'} \,;^{\ell} \rrbracket, \_)\, \phi) \Rightarrow_S ((last(\overrightarrow{s}), \_), \phi \wedge \hat{e})$

If-False

$((\llbracket \texttt{if } (e)\ \overrightarrow{s} \texttt{ else } \overrightarrow{s'} \,;^{\ell} \rrbracket, \_)\, \phi) \Rightarrow_S ((last(\overrightarrow{s'}), \_), \phi \wedge \neg\hat{e})$

Figure 3.11: Transition rules for intra-procedural backward symbolic execution.

in the symbolic configuration in our transition rules for better readability.

Note that *subType* is a predicate in FOL to check the sub-type relation, and *type* and $field$ are functions in FOL. We also use a number of helper functions in our transition rules. The function *pred* returns the previous statement of a given label, and the function *last* returns the last element of a given list. The function *method* returns the method that encloses the statement with the given label. The function *callee* takes as input the label of a call site s, and returns the set of all possible callees. Specifically, it retrieves the signature *sig* of the called method from s, locates the enclosing method M of s in the call graph, and returns the set of all callees of M that match *sig*. The function *callsites* returns the set of all possible call sites of a given method.

Most intra-procedural rules are straightforward. Hence, we next provide more explanations on the inter-procedural rules, which are more interesting. We assume that our language uses the *call-by-value* evaluation strategy. To perform inter-procedural BSE, a context-insensitive call graph is used to guide the execution. The entry point of the call graph is the message handler of the actor. As we execute a method m backwards, there are two possible cases

Method Invocation

$((\llbracket v = v'.m(\overrightarrow{v''})\,;^{\ell}\rrbracket, cs, \hat{v_0}),\ \phi) \Rightarrow_S ((s, cs', \hat{v'}),\ \phi'),$ where

$M = \llbracket C\ m(\overrightarrow{C'\ v'''})\ \{\overrightarrow{s'}\}\rrbracket \qquad M \in callees(\ell) \qquad s = last(\overrightarrow{s'})$

$cs' = cons((pred(\ell), \hat{v}, \hat{v'}),\ cs) \qquad \phi' = \phi \wedge \hat{v_i'''} == \hat{v_i''}$

Return-CallStack Not Empty

$((\llbracket \texttt{return}\ v\,;^{\ell}\rrbracket, cs, \_),\ \phi) \Rightarrow_S ((pred(\ell), cs, \_),\ \phi[\hat{v}/\hat{v'}]),$ where $(\_, \hat{v'}, \_) = car(cs)$

Method Entry-CallStack Not Empty

$((\llbracket \texttt{entry}\,;^{\ell}\rrbracket, cs, \hat{v_0}),\ \phi) \Rightarrow_S (s, cdr(cs), \hat{v_0'}),\ \phi),$ where $(s, \_, \hat{v_0'}) = car(cs)$

Return-CallStack Empty

$((\llbracket \texttt{return}\ v\,;^{\ell}\rrbracket, [\,], \_),\ \phi) \Rightarrow_S ((pred(\ell), [\,], \_),\ \phi)$

Method Entry-CallStack Empty

$((\llbracket \texttt{entry}\,;^{\ell}\rrbracket, [\,], \hat{v_0}),\ \phi) \Rightarrow_S ((pred(\ell'), [\,], \hat{v'}),\ \phi'),$ where

$s = \llbracket v = v'.m(\overrightarrow{v''})\,;^{\ell'}\rrbracket \qquad \ell' \in callsites(M)$

$M = \llbracket C\ m'(\overrightarrow{C'\ v'''})\ \{\overrightarrow{s'}\}\rrbracket = method(\ell') \qquad \phi' = \phi \wedge \hat{v_i'''} == \hat{v_i''}$

Figure 3.12: Transition rules for inter-procedural backward symbolic execution.

regarding the target: 1) the target is outside m, indicating that BSE has previously reached the call site of m and has jumped from that call site to m, and the current call stack must be not empty; 2) the target is inside m, indicating that BSE starts from m and the current call stack must be empty. The first three rules in Figure 3.12 apply to the first case. The *Method Invocation* rule says that upon a method invocation, BSE queries the call graph for all possible callees of the invocation, jumps to the last statement of a possible callee, and adds the constraint that every parameter must be equal to its corresponding argument of the callee (call-by-value). The *Return-CallStack Not Empty* rule says that the variable to which the return value is assigned at the call site is replaced with the return value in the path constraint. The *Method Entry-CallStack Not Empty* rule says that the execution returns to the call site, and the top frame is popped from the call stack. The last two rules in Figure 3.12 apply to the second case. The *Return-CallStack Empty* rule does not update the path constraint, because the caller is unknown at this point, so is the variable that would hold the return value. The *Method Entry-CallStack Empty* says that BSE queries the call graph for all possible callers of the current method, jumps back to a possible call site, and adds the constraint that every argument of the callee are equal to its corresponding parameter in the call site. Note that no constraint over the variable v that holds the return

value is added to the path constraint, because once the execution returns to the call site, it moves backwards and will never use the variable v. The constraints over v do not affect covering the target, and thus need not be added.

3.4.2 Semantics Of Actor Operations In BSE

Figures 3.13 shows the semantics of BSE for actor operations. We put a hat on a symbol to represent a free variable in ϕ. For example, we use $\hat{v}$ in ϕ to represent the corresponding variable v is free. Note that for variables with the same name in different execution contexts, we create distinct variables in ϕ to represent them. The notation $\phi[\hat{v'}/\hat{v}]$ means that every occurrence of $\hat{v}$ in ϕ is syntactically replaced by $\hat{v'}$. It is important to note that whenever such substitutions happen in ϕ, we also perform the corresponding substitutions in the rest of the symbolic configuration. For readability, we omit these subsequent substitutions in our transition rules. We use a number of helper functions in our transition rules. The function *pred* returns the previous statement of a given label, and the function *last* returns the last element of a given list. The function *method* returns the method that encloses the statement with the given label. The function *AC* returns the class name of the actor object mapped by the given actor reference. The function *read* takes as input a free variable representing an object and the field name, and returns the variable representing the field. The function *predCls* takes as input a class name, locates the node of this class in the MFG, finds the predecessors of the node, and returns a set of class name of the predecessors.

The *Actor Creation* rule and the *Message Sending* rule say that upon an actor-creation or message-sending operation, an actor-creation or a message-sending event is added to a pool of pending events μ. Every actor keeps a local time $\hat{t}$, and increases its local time when an actor operation is performed. Hence, the constraint $\hat{t'} < \hat{t}$ indicates that the operation at $\hat{t'}$ *happens before* the operation at t. The *Actor Entry* rules describe potential transitions when BSE reaches the entry of the `onReceive` method of an actor. Reaching the entry of the *onReceive* method implies that this actor must have been created and have received a message. Thus, in both *Actor Entry* rules, a corresponding message request is added to the set Q, indicating that the specific message is required in order for the execution to reach this point, and BSE needs to find an actor that sends the message. There are two possibilities concerning who may create this actor or send a message to this actor. The *Actor Entry-Existing Actor* describes one possibility that this actor is created by an existing actor in α, and the message is also sent from an existing actor; there is no need to introduce new actors in α. The *Actor Entry-New Actor* describes the other possibility: either the actor creation or the message send is done by actors not in α. As a result, a new actor is added to α.

Actor Creation

$\langle \alpha \bullet r \mapsto ((\, [\![v = \texttt{create}(C.\texttt{class}, \overrightarrow{v'});^{\ell}]\!], \_), \hat{t}, \_) \mid \mu \mid \phi \mid \_ \rangle \Rightarrow_S$
$\langle \alpha \bullet r \mapsto ((pred(\ell), \_), \hat{t}', \_) \mid \mu' \mid \phi' \mid \_ \rangle$, where

$\hat{t}', \hat{r'}$ are fresh $\quad \phi' = \phi[\hat{r'}/\hat{v}] \wedge \hat{t}' < \hat{t} \quad \mu' = \mu \cup \{(\hat{r'}, C, \overrightarrow{\hat{v'}}, \hat{t})\}$

Message Sending

$\langle \alpha \bullet r \mapsto ((\, [\![v.\texttt{send}(v');^{\ell}]\!], \_), \hat{t}, \_) \mid \mu \mid \phi \mid \_ \rangle \Rightarrow_S$
$\langle \alpha \bullet r \mapsto ((pred(\ell), \_), \hat{t}', \_) \mid \mu' \mid \phi' \mid \_ \rangle$, where

$\hat{t}'$ is fresh $\quad \phi' = \phi \wedge \hat{t}' < \hat{t} \quad \mu' = \mu \cup \{(\hat{r}, \hat{v}, \hat{v'}, \hat{t})\}$

Actor Entry-Existing Actor

$\langle \alpha \bullet r \mapsto ((\, [\![\texttt{entry}_R;^{\ell}]\!], \_), \hat{t}, Q) \mid \_ \mid \phi \mid \_ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((nil, \_), \hat{t}', Q') \mid \_ \mid \phi' \mid \_ \rangle$

where $\quad \hat{t}'$ is fresh $\quad \phi' = \phi \wedge \hat{t}' < \hat{t}$

$[\![\texttt{void onReceive}(C'v')\{\overrightarrow{C''v''}; \overrightarrow{s}\}]\!] = method(\ell) \quad Q' = Q \cup \{(\hat{v'}, r_{\hat{sender}}, \hat{t})\}$

Actor Entry-New Actor

$\langle \alpha \bullet r \mapsto ((\, [\![\texttt{entry}_R;^{\ell}]\!], \_), \hat{t}, Q) \mid \_ \mid \phi \mid \_ \rangle \Rightarrow_S$
$\langle \alpha \bullet r \mapsto ((nil, \_), \hat{t}', Q') \bullet r' \mapsto ((nil, [\,], \hat{v_0'}), \hat{t''}, [\,]) \mid \_ \mid \phi' \mid \_ \rangle$, where

$\hat{t}', \hat{t''}, \hat{r'}, \hat{v_0'}$ are fresh is fresh $\quad [\![\texttt{void onReceive}(C'v')\{\overrightarrow{C''v''}; \overrightarrow{s}\}]\!] = method(\ell)$

$C \in predCls(AC(r)) \quad Q' = Q \cup \{(\hat{v'}, r_{\hat{sender}}, \hat{t})\} \quad \phi' = \phi \wedge \hat{t}' < \hat{t} \wedge \hat{t''} < \hat{t}$

Messaging Event Matching-Internal

$\langle \alpha \bullet r \mapsto (\_, Q \bullet (\hat{v}, r_{\hat{sender}}, \hat{t})) \mid \mu \bullet (\hat{r'}, \hat{r''}, \hat{v'}, \hat{t'}) \mid \phi \mid \chi \rangle \Rightarrow_S$
$\langle \alpha \bullet r \mapsto (\_, Q) \mid \mu \mid \phi' \mid \chi \rangle$, where

$\phi' = \phi \wedge \hat{r} == \hat{r''} \wedge \hat{v} == \hat{v'} \wedge r_{\hat{sender}} == \hat{r'} \wedge \hat{t}' < \hat{t}$

Messaging Event Matching-External

$\langle \alpha \bullet r \mapsto (\_, Q \bullet (\hat{v}, r_{\hat{sender}}, \hat{t})) \mid \_ \mid \chi \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto (\_, Q) \mid \_ \mid \chi \cup \{(\hat{r}, \hat{v})\} \rangle$

Creation Event Matching

$\langle \alpha \bullet r \mapsto ((\texttt{nil}, \_, \hat{v_0}), \hat{t}, [\,]) \mid \mu \bullet (r', AC(r), \overrightarrow{\hat{v}}, \hat{t'}) \mid \phi \mid \_ \rangle \Rightarrow_S \langle \alpha \mid \mu \mid \phi' \mid \_ \rangle$, where

$\overrightarrow{f} = \mathcal{F}(AC(r)) \quad \phi' = \phi \wedge \hat{r} == \hat{r'} \wedge read(\hat{v_0}, f_i) == \hat{v_i} \wedge \hat{t}' < \hat{t}$

OnReceive Looping

$\langle \alpha \bullet r \mapsto ((\texttt{nil}, \_), \_) \mid \_ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((last(\overrightarrow{s}), \_), \_) \mid \_ \rangle$, where

$[\![\texttt{void onReceive}(C'v')\{\overrightarrow{C''v''}; \overrightarrow{s}\}]\!] = rec(AC(r))$

Self Reference

$\langle \alpha \bullet r \mapsto ((\, [\![v = \texttt{self};^{\ell}]\!], \_), \_) \mid \phi \mid \_ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((pred(\ell), \_), \_) \mid \phi[\hat{r}/\hat{v}] \mid \_ \rangle$

Sender Reference

$\langle \alpha \bullet r \mapsto ((\, [\![v = \texttt{sender};^{\ell}]\!], \_), \_) \mid \phi \mid \_ \rangle \Rightarrow_S \langle \alpha \bullet r \mapsto ((prev(\ell), \_), \_) \mid \phi[r_{\hat{sender}}/\hat{v}] \mid \_ \rangle$

Figure 3.13: Transition rules for actor operations in backward symbolic execution.

The MFG is queried to obtain the predecessors of this actor class, which is the set of actor classes that may create or send a message to this actor. Then an actor with the default initial state is created in α with its type being one of the predecessors. This is the only rule that introduces new actors to our exploration.

It is important to note that jumping between actors is different from jumping between methods. In inter-procedural BSE, the execution directly jump to call site after reaching the entry point of the callee. By starting from the call site, the BSE ignores the computations after the call site, which is safe in sequential programs because the computations are guaranteed to happen after the call. However, in the actor case, the execution has to jump the last statement of the `onReceive` method instead of directly to message-sending site in the new actor. The reason is that messages sent after this site may arrive before the message sent from this site.

A message request is fulfilled either by a pending message event in μ sent from an actor inside the system or, if the actor is a receptionist, by a message sent from the external environment. The *Messaging Event Matching-Internal* rule describes the first case, in which the matched request and event are remove from Q and μ respectively, and a happens-before constraint between the message receive and send operations is added to ϕ. The *Messaging Event Matching-External* rule describes the second case, in which the request is removed from Q, and an external message is added to χ. The *Creation Event Matching* rule says that a pending actor-creation event is matched with an actor in α. Note that to match a creation event, the type of the actor must be the same as the type specified in the creation event, and the message request set Q of the actor must be empty, indicating all message requests are fulfilled. The *Receive Looping* rule says that an idle actor can start an execution from the exit of the *onReceive* method.

Recall our bank example from Figure 3.6, assume that our target is to violate the assertion at line 30 in the `Server` class. Due to space constraints, we only walk BSE through a successful branch that leads to a test case being generated. BSE starts from line 30 and executes backwards until it reaches the entry of the `onReceive` method, adding a *WithdrawMessage* request to its Q. BSE may query the MFG and create actors for each potential actor that can send messages to the current one. However, we favor the exploration of the branch that doesn't create more actors. The client and server actors at this point match the *Receive Looping* rule. We execute the rule twice for the client (a *WithdrawMessage* and a *DepositMessage* are added to its Q) and once for the server (a *DepositMessage* is added to its Q). The code of the `onReceive` of the client will generate two messages (a *DepositMessage* and a *WithdrawMessage*) to be added to the μ.Then, the client's message requests get matched by external messages since it's a receptionist, and those of the server get matched

against the ones sent by the client in the μ. The path constraint at this point dictates that the balance should be bigger than the withdraw amount, when it occurs in the client. Therefore if the client first receives a deposit of 50$, and then a withdraw of 120$, the constraint is satisfied. On the server side, if the server receives the withdraw of 120$ first before the 50$ deposit, the assertion on line 30 is reached and violated. The path constraint reflects the order of sends and receives making the test case posssible.

3.4.3 Path Exploration In BSE

The initial symbolic configuration is that the actor map α contains only one actor with the statement being the target, and the event pool μ contains the actor-creation events from the external environment. BSE starts with the initial configuration and takes one transition at a step. The computation branches when multiple transition rules can be matched on one configuration. BSE uses the depth-first search strategy for path exploration. At each branching point, we pick one transition from all enabled transitions, and check if the path constraints in the new configuration is satisfiable. If satisfiable, we continue the exploration on the new configuration; otherwise, we backtrack. The final accepting configurations are the ones with α being empty and ϕ being satisfiable. A system test can be constructed from the model of ϕ, the transition path, and the set of external messages.

Because actors in the configuration proceed their computations concurrently, almost any configuration has multiple enabled transitions. As a result, the search space in BSE is intractable. To address this problem, we propose two search heuristics and a feedback-directed search technique to efficiently find a feasible path in the huge search space.

Search Heuristics

Our first heuristic is that BSE always explores a message handler atomically. In other words, once BSE starts a transition of local computations in a message handler of an actor, all transitions on other actors are disabled and BSE will keep exploring this message handler until reaching the entry of the message handler. As a result, the number of enabled transitions on each symbolic configuration is reduced. This heuristic leverages the atomicity of the macro-step semantics [2] in the Actor model–messages to a given actor are processed one at a time without interleaving. Macro-step is also enabled by the fact that the concurrent execution of message handlers on different actors need not be interleaved (i.e., messages to different actors can be sequentialized). This is because actors do not share states. Therefore, the heuristic is safe: it reduces the search space in BSE without missing any tests that can

potentially cover the target.

Our second heuristic keeps the number of actors in the generated test small in order to avoid exploring unnecessary paths. This heuristic is based on the conjecture that most concurrency bugs may be triggered by considering interactions of a small number of actors. The conjecture is the result of a previous finding that most concurrency bugs in multi-threaded programs can be triggered using two threads [40]. With this conjecture, we assign different weights to transition rules for actor operations. When multiple transition rules are enabled on a configuration, the probability of picking a rule is based on its weight (rules with more weights have a higher chance of being picked). We give a much lower weight to the $ActorEntry - NewActor$ rule, which is the only rule to introduce new actors to a test. This is because introducing a new actor opens up a whole new search space – BSE has to find a feasible execution trace on this actor. In this way, we keep the number of actors in our test small, and avoid fruitless explorations. In addition, we give more weights to transition rules that consume pending events in the event pool μ so that message requests from actors can be fulfilled as soon as possible. Recall that a test is generated only when BSE reaches a final accepting configuration, where the actor map α must be empty. An actor is removed from α only when all of its message requests are fulfilled. Hence, fulfilling these message requests helps BSE find a test efficiently.

Feedback-Directed Search

Heuristics do not always work well. There are cases where a large number of transitions are enabled, but only a few of them can lead to a feasible path. If the heuristics do not bias towards these transitions, BSE will frequently hit infeasible paths. The feedback-directed search technique guides BSE out from such undesirable situations by leveraging the unsatisfiable cores of the path constraint from the previous infeasible paths. An unsatisfiable core is a subset of clauses in the original constraint such that the conjunction of these clauses is unsatisfiable. To make the path constraint feasible, the clauses in the unsatisfiable core need to be changed. The idea of our feedback-directed technique is to drive the execution towards the code that changes the values of the variables in the unsatisfiable core, hoping that the changes will make the path constraint satisfiable.

Our feedback-directed technique has two steps. In the first step, we identify a set of code instructions that can potentially change the unsatisfiable core. We obtain the unsatisfiable core of the path constraint directly from the underlying SMT solver Z3 [65]. Then we extract all the variables from the unsatisfiable core, and map these variables to corresponding program variables. This can be done without additional overhead, because our symbolic

```
1  private int pingsLeft = 100;
2  public void onReceive (Object message) {
3      if(message instanceof PongMessage) {
4          pongActor.tell(new PingMessage(), getSelf());
5          pingsLeft --;
6          if(pingsLeft == 0) {
7              \\ target
8          } ...
9      } ...
10 }
```

Figure 3.14: An example from our subjects that illustrates the feedback-directed search technique.

execution keeps track of the mapping between the variables in path constraints and program variables. For each program variable, we identify a set of instructions that define this variable (definition sites). In our implementation, BSE is performed on an IR that is in the static-single-assignment form. Hence, there is only one definition site per variable. In the second step, we drive the execution to the definition sites identified in the first step. To do this, we compute the transitions that may lead to at least one of these definition sites. A transition may lead to a definition site if the statement transited to is reachable from the definition site in the inter-procedural control flow graph. We prioritize these transitions over the others.

Figure 3.14 shows the message handler of the ping actor in the Ping-Pong example. The `pingsLeft` field keeps track of the ping messages sent out, and is initially set to 100. To cover the target at Line 7, the ping actor has to receive 100 pong messages. Suppose that when BSE first reaches the entry of the message handler from the target, it chooses to jump to the constructor of the ping actor, meaning that only one pong message is received after creating this actor. Obviously, this path is infeasible. Its path constraint is $p = 100 \wedge p - 1 = 0 \wedge subType(type(m), PongMessage)$, where p and m map to the program variables `pingsLeft` and `message`, respectively. The unsatisfiable core of this path constraint is $\{p = 100, p - 1 = 0\}$, whose only variable maps to `pingsLeft`. Thus, Line 1 and Line 5 are identified as the definition sites for `pingsLeft`. Then BSE backtracks to the entry of the message handler, and picks the transition that jumps to Line 8, because it may lead to the definition site at Line 5. This transition indicates that the ping actor has received two pong messages. Note that BSE does not pick the transition that may lead to Line 1 (i.e., the transition that jumps to the constructor), because it has been explored previously, leading to an infeasible path. This process iterates 100 times and BSE finds a feasible path in which

the ping actor receives 100 pong messages. Without this technique, in each iteration, BSE may try other messages that do not affect `pingsLeft`, thus making the search inefficient.

Subjects	**LOC**	**Description**
Micro Bench.	50 - 200	8 well-known actor example programs
Concurrency Bench.	100 - 400	8 classic concurrency problems
Parallelism Bench.	200 - 1,000	14 realistic parallel applications
AkkaCrawler	715	A web crawler and indexer
Batch	1,309	A concurrent batch processing framework
Parallec	12,457	A parallel client firing requests and aggregating responses
Stone	20,935	An online game server framework

Table 3.1: Characteristics of the subjects in our evaluations

3.5 IMPLEMENTATION AND EVALUATION

3.5.1 TAP Implementation

We implement our method in a tool called TAP for actor systems developed with Java *Akka*. TAP is built on top of WALA [66], a static analysis infrastructure for Java. TAP transforms the Java bytecode of the system under test to WALA IR and performs analysis on WALA IR. The benefit of working on WALA IR is that one can directly use the basic built-in analyses provided by WALA. TAP uses multiple WALA built-in analyses such as class hierarchy analysis, call graph analysis, and points-to analysis. Since Scala *Akka* programs are also compiled to Java bytecode, TAP in principle may be used to analyze Scala *Akka* programs as well. However, Scala *Akka* has a different set of interfaces, and substantial engineering work is required to support Scala *Akka*. We plan to support Scala *Akka* in the future.

TAP consists of two major components, an MFG builder containing ∼4,000 lines of Java code and a BSE engine containing ∼11,000 lines of Java code. The implementation of the MFG builder closely follows the formalizations and the iterative MFG construction algorithm described in Section 3.3. A key part for MFG construction is resolving recipients and messages in message-sending sites. TAP maintains a map from an actor reference to a set of actor objects that are possibly referenced by it. This map is used to resolve `ActorRef` pointers. TAP queries WALA's points-to analysis to resolve all other pointers.

The BSE engine includes a backward symbolic interpreter on WALA IR as well as the search techniques. The interpreter implements a transition rule (similar to the semantic

rules in our BSE formalization) for each type of statements in WALA IR. The actor library calls are interpreted using our semantic models so that TAP does not explore the actor library methods. The BSE engine forks a new symbolic configuration whenever the computation branches. TAP uses Z3 [65] as the off-the-shelf SMT solver for solving path constraints and computing unsatisfiable cores. An important deviation from the formalizations is that TAP implements the FIFO message delivery semantics, because our target actor framework, Java *Akka* guarantees the FIFO semantics. To implement the FIFO semantics, TAP models the pending messages as a set of lists rather than a multi-set. Each list models a FIFO communication channel between a pair of actors so that the message sending order is preserved.

3.5.2 Evaluation

We evaluate TAP on a set of third-party benchmarks called *Savina* [41] as well as four randomly selected open-source projects from GitHub. Our experiments consist of two parts: 1) the evaluation on the MFG construction analysis, measuring the size of the MFGs, analysis time, and the effectiveness of the optimizations; 2) the evaluation on the effectiveness of our test generation method.

Subjects	**Baseline Analysis**				**Optimized Analysis**			
	# Nodes	# Edges	# Labels	Time (s)	# Nodes	# Edges	# Labels	Time (s)
Micro	2.5	4.3	6.5	45	2.5	4.3	6.2	45
Concurrency	3.8	10.4	16.5	56	3.8	9.3	14.4	59
Parallelism	4.5	17.5	24.9	79	4.5	**15.8**	**19.2**	**72**
AkkaCrawler	3	6	15	57	3	6	12	55
Batch	5	12	31	85	5	**10**	**21**	**77**
Parallec	8	16	67	190	8	**13**	**46**	**131**
Stone	38	74	173	243	38	**58**	**121**	**169**

Table 3.2: Comparison between the baseline MFG analysis and the optimized MFG analysis. The numbers for the three benchmark categories are averages.

Table 3.1 describes the subjects used in our evaluation. The *Savina* benchmarks consist of 30 diverse programs written purely using actors. *Savina* has three categories: micro benchmarks with 8 well-known actor examples, concurrency benchmarks with 8 classic concurrency problems, and parallel benchmarks with 14 realistic parallel applications. *Savina* has been used in the actor community for various evaluation purposes, such as performance comparison of actor languages/frameworks [41, 67], actor profiling [68], and mapping from message passing concurrency to threads [69]. The original *Savina* does not have a Java *Akka* implementation. We transformed the Scala *Akka* implementation in *Savina* into Java *Akka*

and used the transformed version in our experiment because TAP currently supports only Java *Akka*. We had at least two actor programmers double check that the transformed Java version is equivalent to the Scala version.

All four open source projects are written in Java using the Java *Akka* library. Most of their application logic is implemented in actors. AkkaCrawler is a parallel web crawler and indexer. Batch is a framework for concurrent batch processing. Parallec is a scalable asynchronous client, developed by eBay, for firing large numbers of HTTP/SSH/TCP/UDP requests and aggregating responses in parallel. Stone is a framework for developing online game servers. From all the actor-based Java *Akka* projects that we can find on Github, Parallec and Stone are among the largest projects. Some projects mix the Actor model with other concurrency models [70]. We exclude those projects from our evaluation because TAP does not handle other concurrency models such as threads. All our experiments ran on a quad-core machine with 16 GB of RAM, running a 64-bit Ubuntu 14 system.

3.5.3 Results on MFG Construction

To demonstrate the effectiveness of the optimizations described in Section 3.3.3, we compare the optimized MFG analysis to the one without optimizations in terms of the size of the MFGs and the time taken for MFG construction. We measure the size of an MFG using the number of nodes, the number of edges and the number of labels on all edges. Overall, 92% of the `onReceive` methods in our subjects match the code pattern for optimizations (i.e., the message handler has a top-level *if* statement that checks for the message type).

MFG Size. Table 3.2 shows the comparison results. The numbers for the three benchmark categories are averages because there are multiple projects in each category. On average, the optimized analysis reduces the number of edges by 11% and the number of labels by 23%. The number of nodes is not reduced because our analysis creates only one node per actor class. Recall that our optimizations are safe, indicating that all the reduced edges and labels are false positives. The results show that our optimizations substantially improve the precision of MFG analysis.

The results also show that the optimized analysis reduces a far larger percentage of edges and labels on larger projects. Table 3.2 highlights (in bold) cases where our optimizations significantly reduces the size of MFGs. For instance, the optimized analysis reduces edges by 19% and labels by 31% for the Parallec project, and reduces edges by 22% and labels by 30% for the Stone project. However, on small subjects such as the micro benchmarks, our optimizations do not produce a significant difference. The reason is that the computed points-to sets in larger projects are typically larger than those computed in smaller projects.

Our optimizations often reduces the points-to set to only one element or a much smaller subset in a top-level branch. Therefore, the larger the points-to sets are, the more false positives are reduced. In summary, the optimized analysis has a bigger impact on larger projects.

Analysis Time. We ran the same experiment five times to obtain the average time taken by each analysis on each subject. An interesting observation is that the optimized analysis takes much less time than the baseline analysis does in projects where the optimized analysis reduces the MFG size significantly. For instance, on both Parallec and Stone projects, the analysis time drops about 30% with the optimizations. In other words, the optimized analysis produces more precise results with less time. Our investigation indicates that with smaller points-to sets, the iterative MFG construction algorithm reaches the fixed point faster: having larger points-to sets implies more candidate actors or messages, and this often leads to more iterations for the algorithm to converge. The overhead of our optimizations is negligible, because the optimized analysis performs only a simple structural check on the control flow graph of the `onReceive` method. As shown in the results, the two analyses take similar time on small projects such as the micro benchmarks and the AkkaCrawler project.

3.5.4 Results on Test Generation

To evaluate the effectiveness of our test generation method, we randomly selected basic blocks in actor classes as targets from all subjects, and for each target, we applied Tap to generate tests to cover it. To avoid biases, we evenly distributed the targets based on the size of actor classes in each project. In practice, the targets may be software patches [57], assertions, and suspicious code locations. In total, we selected 500 targets for the *Savina* benchmarks and 500 targets for the four open source projects. The effectiveness of our method is measured by the percentage of targets covered. A target is covered only when Tap finds a feasible path to the target within the given timeout.

Our problem settings require the specification of receptionists for each actor system. Unfortunately, such information is not specified in our subjects. Therefore, we manually inferred the receptionists for each project from its drivers and tests. We set a timeout of ten minutes per target excluding the time for MFG construction. To compare our search techniques, we ran Tap using the following five settings: 1) **Random**, pick a transition randomly from all matched rules on a symbolic configuration; 2) **H1**, enable only the first heuristic; 3) **H2**, enable only the second heuristic; 4) **H1 + H2**, enable both heuristics; 5) **H + F**, enable both heuristics and the feedback-directed technique. All five settings used the depth-first search strategy.

Subjects	Targets	Random	H1	H2	H1 + H2	H + F
		Cov (%)	Cov (%)	Cov (%)	Cov (%)	Cov (%)
Micro	97	54%	57%	61%	78%	85%
Concurrency	162	45%	56%	49%	70%	77%
Parallelism	241	36%	43%	59%	67%	72%
AkkaCrawler	39	54%	64%	72%	87%	90%
Batch	60	63%	72%	70%	85%	92%
Parallec	178	42%	46%	48%	51%	78%
Stone	223	29%	43%	48%	56%	75%
Total	1000	41%	49%	54%	65%	78%

Table 3.3: The target coverage results of running TAP with five settings.

Target Coverage

Table 3.3 summarizes the results of running TAP with the five settings. Column 2 shows the number of targets selected for each subject. Columns 3-7 show the number and the percentage of the targets covered by the five settings, respectively. The last row shows the average time (in seconds) taken for covering a target in each setting excluding the time for MFG construction. Overall, the combination of heuristics and feedback-directed technique is effective in covering targets. Search heuristics increase the target coverage from 41% to 65%. The feedback-directed technique further increases the target coverage to 78%.

The Random setting does not work well. It times out in 228 out of 1000 cases. The major problem with Random is that it often introduces many unnecessary actors to path exploration. Introducing a new actor in a test is an expensive operation, because it opens up additional search space for TAP to find a feasible execution trace on the new actor. As a result, Random wastes lots of resources exploring traces for unnecessary actors, and takes longer time to cover a target. In additional, the tests generated by Random are typically larger in terms of the number of actors. The H1 setting suffers the same problem. However, it reduces the search space by sequentializing the execution of message handlers. As a result, the number of enabled transitions on each symbolic configuration in H1 is much smaller than that in Random. Due to the space reduction, H1 improves the target coverage to 49%.

The H2 setting improves Random by keeping the tests as small as possible to avoid exploring unnecessary space. Our experiment results show that in many cases, the target can be reached with no more than three actors. For example, many subjects use the master-worker pattern to implement parallelism. The workers proceed in parallel, and do not interact with each other. In such cases, it suffices to cover any target in the worker with only two actors: one master and one worker. Creating new workers only adds complexity to the problem. H2

is very efficient in covering such targets because it assigns a very low weight to transitions that introduce new actors.

The feedback-directed technique is particularly useful when our heuristics do not work well and BSE frequently hits infeasible paths. In our experiment, we find that there are a number of cases where covering the target requires creating multiple actors of the same class (e.g., comparing the IDs of actors). In these cases, the heuristics work poorly because they prefer to reuse the existing actor rather than create a new actor of the same class. As a result, the heuristics keep hitting infeasible paths in these cases. The feedback-directed technique is quite effective in guiding BSE to find a feasible path. For instance, in the case of checking for different IDs, it directly identifies that the ID field of the actor needs to be changed, because the unsatisfiable core contains variables that map to this field. Since the only way to change the ID field of the actor is through its constructor, the feedback-directed technique prioritizes the transitions that introduce new actors to be explored first, and thus quickly finds a feasible path.

We analyze the cases in which TAP fails to cover the targets in the H + F setting. More than half of the cases are due to a lack of environment modeling (e.g., access to database and network). Such issues can be mitigated by adding models for calls to the environment. The rest of the cases are mainly due to timeouts for the exploration and complex constraints that Z3 fails to solve.

Bug Detection

By running TAP to cover these 1,000 targets, we are able to find six distinct bugs in our subjects. All six bugs are found in the *Savina* benchmarks in three projects. Five out of the six bugs are crash bugs. One bug is less critical: a non-crash warning from *Akka* regarding messages sent to actors that have been killed. We have confirmed that all bugs are triggered in both the original benchmarks and the transformed versions with our generated tests. We diagnose the six bugs and find that all five crash bugs are caused by out-of-order message delivery. Such bugs are hard to reveal locally because out-of-order message delivery is unlikely to happen locally. The other bug is caused by sending two stop messages to kill an actor, and the recipient actor kills itself after receiving the first stop message.

Figure 3.15 shows one crash bug found in the ThreadRing benchmark. There is a potential null de-reference on the `nextActor` field at Line 6. The ThreadRing system starts with a coordinator sending a DataMessage to each token passer to inform them the next passer and form a ring among them. The coordinator then sends a token to one passer in the ring, and then the token is passed from one passer to another in the ring. The passer sets its `nextActor`

```
1  public void onReceive (Object message) {
2      if (message instanceof TokenMessage) {
3          TokenMessage token = (TokenMessage)message;
4          if(token.hasNext()) {
5              // bug: potential null de-reference on nextActor
6              this.nextActor.tell(token.next(), getSelf());
7          } ...
8      } else if (message instanceof DataMessage) {
9          this.nextActor = (ActorRef) ((DataMessage) message).data;
10     } ...
11 }
```

Figure 3.15: A bug caused by out-of-order message delivery in the ThreadRing benchmark

field at Line 9 upon receiving a `DataMessage` and sends the token at Line 6 upon receiving a `TokenMessage`. The assumption is that every passer must set the `nextActor` before sending the token (i.e., receive the `DataMessage` before the `TokenMessage`). Since the *Akka* framework guarantees FIFO message delivery, this assumption holds for the first passer. However, the assumption may not hold for the other passers. It is possible that the second passer receives the `TokenMessage` from the first passer before receiving the `DataMessage` from the coordinator. Although the `DataMessage` is sent before the `TokenMessage`, the two messages are sent by different senders, and may be delivered out of order. In this case, a null pointer exception is thrown in the second passer. TAP found this bug because the exceptional branch of Line 6 (WALA IR contains exceptional branches for potential null dereferences) happened to be chosen as a target. A simple fix to this bug is adding a null check on `nextActor` before passing the token.

CHAPTER 4: ACTOR SPECIFICATION DIAGRAM INFERENCE

The task of understanding and reasoning about the behaviors of concurrent systems is challenging. The non-determinism in concurrent systems can result in a large number of potential behaviors, making it impractical to explore every possible system behavior. Unfortunately, developers are forced to face the challenge of understanding system behaviors in many real world scenarios, such as when diagnosing unintended system behaviors, making changes to legacy code, or writing test oracles. A common approach to understand the behavior of a system is to manually read its code or examine the execution traces, because systems in practice often lack formal behavioral specifications. However, large and complex software systems often exceed the developer's ability to navigate and reason.

Various automated techniques [71, 72, 73, 74, 75, 76] have been proposed to construct behavioral models of concurrent systems such as UML Sequence Diagram [77], Message Sequence Chart [78], Message Sequence Graph [79, 80], and Communicating Finite-State Machine [81] to facilitate understanding concurrent programs. These models are often simplified by omitting implementation details and focusing on the communications between concurrent processes. Thus, the models make it easier for developers to understand a system's behaviors holistically. Applications of these models include software comprehension, bug detection, model-based testing, and verification.

A common approach towards this problem is to infer such models from system logs or execution traces. Previous work [73, 74, 75] applies various generalization techniques to construct a model out of traces from concrete executions. Then the generalized model is further refined through abstract-refinement techniques such as CEGAR [82], so that the refined model accepts all the execution traces. The foundation of all these techniques is the dynamic information obtained from the execution traces. One limitation of these techniques is that the execution traces often do not cover all the system behaviors. Therefore, on one hand, the inferred model can miss certain behaviors even with generalization. On the other hand, the inferred model can include invalid system properties, because the execution traces that falsify these properties are not collected. An alternative is to apply computational (or algorithmic) learning to traces to build a model of the system [83, 84]. The inferred model can be finite state automata communicating with ordered messages (FIFO channels between pairs of automata). Such inference is done by generalizing observed traces. However, the learning process requires a specification to generate negative examples or feedback for the learning process.

In this chapter, we present a method for inferring specification diagrams [85] for actor

systems; specification diagrams rigorously describe the global behavior of an actor system in terms of actor operations such as message send and receive. To address limited information in available traces, we propose a hybrid approach that combines static analysis with dynamic analysis. We first use static analysis to construct an abstract specification diagram. Compared to the generalization from concrete executions, the advantage of using static analysis is that it can soundly capture all possible behaviors of a system as well as reliably establish the temporal orders between events from control flows and causal relationship of message send and receive. While static analysis constructs the specification diagram in a sound manner, it may include events that can never happen in concrete executions (false positives). In addition, it is difficult for static analysis to infer the exact number of loop iterations as well as coordination constraints encoded in actor states. Hence, we further refine the abstract specification diagram using dynamic information from concrete execution traces. In particular, we dynamically detect *likely invariants* [42] of the actor system, and use them to infer coordination constraints as well as reduce false positives in the specification diagram.

It is important to note that we assume First-In-First-Out (FIFO) message delivery between two actors in this chapter, because many concurrent programming models including our target actor framework *Akka*, guarantee FIFO message delivery. In practice, it would be difficult to program against models without FIFO order guarantee.

The rest of this chapter is organized as follows. We first provide preliminaries on the actor specification diagram and a comparison between the actor specification diagram and other formal behavioral models. Then we describe our inference method in detail including a static analysis to construct an abstract specification diagram and a dynamic analysis to refine the abstract diagram through likely invariants. Finally, we describe the implementation of our inference method, present and discuss the evaluation results.

4.1 PRELIMINARIES

In this section, we provide a short introduction to the actor specification diagram, including its diagrammatic and textual representations. We also present an example of the actor specification diagram for a real-world actor program. We then compare the actor specification diagram with other formal behavioral models of concurrent systems, such as message sequence chart, message sequence graph, communicating finite-state machine, and multiparty session types.

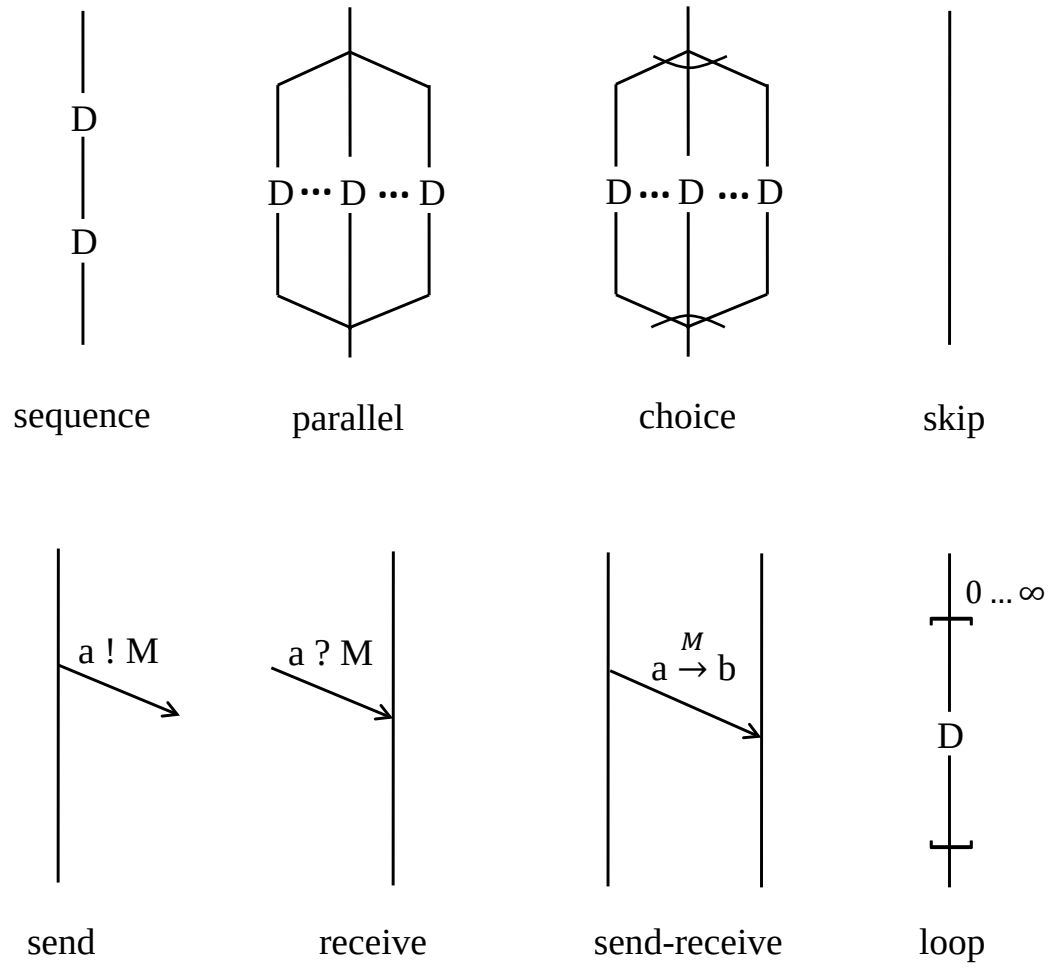


Figure 4.1: Specification diagram constructs

4.1.1 Specification Diagram for Actor Systems

The specification diagram [85] for actor systems is a novel form of graphical notation for specifying open distributed object systems. It is designed for describing message-passing behaviors, with expressiveness, understandability, as well as formal semantic underpinnings. The specification diagram differs from existing actor and process algebra in that it gives a graphical specification.

We next describe the syntax and the informal meanings of the specification diagram (see [85] for a formal definition on the semantics of the specification diagram). There are two forms of the specification diagram: the graphical one and the textual one. The graphical notation is more intuitive and thus intended for use in practice such as system design. The textual notation is more concise and thus intended for mathematical study.

Figure 4.1 shows a subset of constructs of the specification diagram. We show only the constructs that specify the control flows and actor-related operations. The full syntax of the specification diagram includes constructs for manipulating actor local states such as assignment, variable declaration, and assertion. We omit these constructs because our goal is to infer a high level communication model with simplicity and good understandability. In fact, we infer a restricted version of the specification diagram, which contains only the control flows and the events from actor operations such as message send and receive.

The notation D in the constructs represents a self-contained specification diagram. The vertical lines in the specification diagram are called *casual thread*. A casual thread indicates progress in time going down. It expresses the abstract causal orders on events, with the events above necessarily leading to the events below. For the purpose of specifying the temporal orders between events, we augment the semantics of the causal thread to express the happens-before orders so that the events above happen before the event below. The causal order is a subset of the happens-before order because any event that causes another event must happen before the other event. Hence, we call these vertical lines *time thread* to better reflect their meanings.

Note that there is no necessary connection between these time threads and actors. A time thread indicates only the temporal orders of events on this thread, and nothing more. It is possible that a single time thread contains events from multiple actors. On the other hand, events from a single actor may be placed on a single time thread or on multiple time threads.

We next describe the textual notation and the informal semantics of each construct for the specification diagram.

- **sequence** ($D_1 \,;\, D_2$) Events in diagram D_1 happen before events in diagram D_2.
- **parallel** ($D_1 \mid D_2$) Events in parallel diagrams D_1 and D_2 have no specified temporal orders between them, but happen after the events above and before the events below.
- **choice** ($D_1 \oplus D_2$) Either diagram D_1 or diagram D_2 is taken. Fairness is not guaranteed in the sense that the same branch could always be taken for a particular computation. Note that there is no temporal order between events in these diagrams.
- **skip** (*skip*) Inaction
- **loop** ($[D]^{0\cdots\infty}$) Diagram D is iterated n times, where the integer n ranges from 0 to ∞.
- **send** ($a \,!\, M$) Actor a sends a message with content M.

- **receive** ($a \,?\, M$) Actor a receives a message with content M.
- **send-receive** ($a \xrightarrow{M} b$) Actor a sends a message with content M to actor b, and actor b receives the message. This notation indicates that the message send event happens before the message receive event because of the causal order.

4.1.2 Example

We next illustrate how these constructs can be used to build a specification diagram for a real-world program. Figure 4.2 shows the code of the simplified pingpong actors from the *Savina* benchmark. The main method first creates an actor system as well as a pong actor and a ping actor in this system (lines 2-4). More precisely, the `actorOf` method creates an actor object and returns an actor reference `ActorRef`; the actor object is mapped to the actor reference. Note that the actor reference is not the actor object. With an actor reference, one can only send messages to its corresponding actor, but cannot access the fields and member functions of this actor. Thus, the local state of an actor is encapsulated. At line 4, a configuration parameter `N` and the reference of the pong actor are passed to the constructor of the ping actor via the `props` method, to initialize the `pingsLeft` field and the `pongActor` field (line 9), respectively. The main method then sends a `Start` message to the ping actor via the `tell` method (line 5). The `tell` method does asynchronous message passing. The receiver object of the method is the reference of the recipient actor, the first argument is the message being sent, and the second argument is the reference of the sender actor.

Upon receiving the `Start` message, the `onReceive` method of the ping actor is called. The ping actor sends a `Ping` message to the pong actor (line 13), and decreases its `pingsLeft` field. The `Ping` message further triggers the pong actor to send back a `Pong` message (line 29). The `getSelf` method and the `getSender` method return respectively the reference of "this" actor and the reference of the message sender. The loop continues until the `pingsLeft` field decreases to zero. Then the ping actor sends a `Stop` message to the pong actor, and kills itself via the `stop` method (line 17). After receiving the `Stop` message, the pong actor also kills itself (line 31), and the conversation between these two actors ends.

The corresponding specification diagram for the pingpong example is shown in Figure 4.3. For better readability, we use p and q as the shorthands for the ping actor and the pong actor. The specification diagram starts with the ping actor receiving an external `start` message. It is then followed by a subdiagram consisting of a loop construct. The loop body is a diagram with a parallel construct, which consists of two sequenced *send-receive* operations: the ping actor sends a ping message to the pong actor, and the pong actor receives it; then the pong

```
1  public static void main(String[] args) {
2    ActorSystem system = ActorSystem.create("pingpong");
3    ActorRef pongActor = system.actorOf(PongActor.props());
4    ActorRef pingActor = system.actorOf(PingActor.props(N, pongActor));
5    pingActor.tell(new Start(), null);
6  }
7
8  public class PingActor extends UntypedActor {
9         ActoRef pongActor; int pingsLeft;
10        public PingActor(int pingsLeft, ActorRef pongActor) {
11                this.pingsLeft = pingsLeft;
12                this.pongActor = pongActor;
13        }
14        public void onReceive(Object msg) {
15                if (msg instanceof Pong) {
16                        if (pingsLeft > 0) {
17                                pongActor.tell(new Ping(), getSelf());
18                                pingsLeft -= 1;
19                        } else {
20                                pongActor.tell(new Stop(), getSelf());
21                                getContext().stop(getSelf());
22                        }
23                } else if (msg instanceof Start) {
24                        pongActor.tell(new Ping(), getSelf());
25                        pingsLeft -= 1;
26                }
27        }
28 }
29
30 public class PongActor extends UntypedActor {
31        public void onReceive(Object msg) {
32                if (msg instanceof Ping) {
33                        getSender().tell(new Pong(), getSelf());
34                } else if (msg instanceof Stop) {
35                        getContext().stop(getSelf());
36                }
37        }
38 }
```

Figure 4.2: A simplified example of pingpong actors

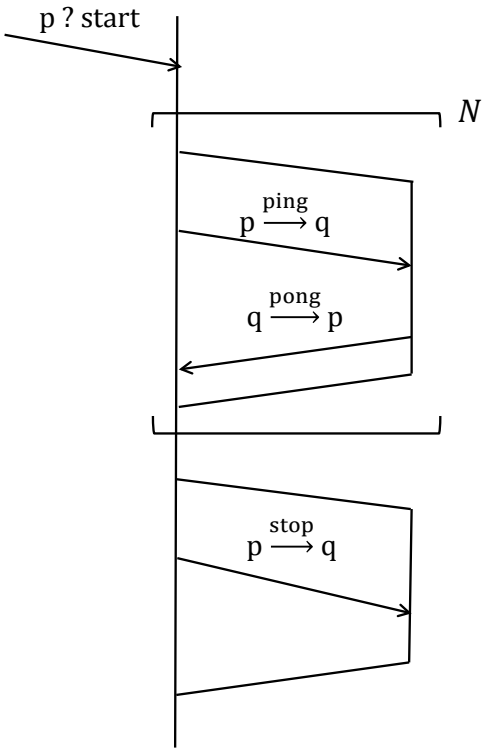


Figure 4.3: Specification diagram for the pingpong example

actor sends a pong message to the ping actor, and the ping actor receives it. This back and forth message exchange repeats exactly N times. Note that the number of iterations comes from the configuration variable `N` in the program. After the loop, the ping actor sends a stop message to the pong actor, and the pong actor receives the stop message.

As shown in the example, the specification diagram is able to describe the global behaviors of the system in a simple and concise manner, with just a few constructs. The interactions between actors are explicitly described in the specification diagram. For example, the loop clearly depicts the repeated message exchanges between the ping pong actors In contrast, the actual implementation in Figure 4.2 does not contain explicit loops. Developers need to reason about the entire code to discover the implicit loop between the two actors.

In addition, the temporal orders between the events are precisely and explicitly reflected by the specification diagram. The receive event of the start message on the ping actor happens before all events in the loop, the send and the receive of the stop message happen after all events in the loop. Within the loop, the send event of the ping message is followed by the receive event of the ping message, followed by the send event of the pong message, finally followed by the receive event of the pong message. Again, in contrast, the code of the example does not explicitly express any order between these events.

4.1.3 Other Behavioral Models

A number of other models have been proposed to specify the behavior of concurrent systems. We briefly introduce some of the popular models and compare them with actor specification diagram.

Message Sequence Chart and Message Sequence Graph

Message Sequence Chart (MSC) [78] is a graphical notation for describing message exchanges between concurrent processes. Similar to actor specification diagram, it consists of vertical lines and arrows across vertical lines. Each vertical line indicates that time progresses as the line goes down. Each arrow represents a message exchange, and the event of sending the message precedes the event of receiving the message.

However, a key difference between actor specification diagram and MSC is the semantics of the vertical lines. In MSC, each vertical line represents exactly one concurrent process. Given that time progresses as the vertical line goes down, the temporal order of the events on the vertical line is a total order. That is, events on each concurrent process must be totally ordered by time. As a result, one MSC can only specify a single execution scenario. Due to the nondeterminism in the arrival order of messages on concurrent processes, multiple MSCs would be required to capture all possible execution scenarios of a concurrent system. In the real-world word, a large number of MSCs is often needed for a non-trivial system. In contrast, the vertical line in actor specification diagram does not represent an actor or a concurrent process. It simply represents the time progress, indicating temporal orders between events. A vertical line does not necessarily correspond to one concurrent process. This allows actor specification diagram to depict multiple execution scenarios in one diagram, because the events of one actor can be placed on multiple vertical lines. If the temporal order between two events of an actor is nondeterministic, the two events can be placed on two parallel vertical lines in one diagram instead of creating two diagrams for both possibilities.

Given that a concurrent system often has a large number of MSCs, Message Sequence Chart-Graph or Message Sequence Graph (MSG) has been proposed to represent a collection of MSCs in a compact graph [79, 80]. MSG is a regular composition (choice, concatenation, and repetition) of MSCs. Formally, an MSG is a directed graph $(V, E, V_s, V_f, \lambda)$, in which V is the set of vertices, E is the set of edges, V_s is a set of entry vertices, V_f is a set of accepting vertices, and λ is a labeling function that maps every vertex to an MSC. For any path $(v_1, v_2, \ldots, v_n)$, where $v_1 \in V_s \wedge v_n \in V_f$, the single MSC derived by concatenating basic MSCs $\lambda(v_1) \circ \lambda(v_2) \circ \cdots \circ \lambda(v_n)$, represents a valid execution scenario of the system. Actor

specification diagram is different from MSG in that there is no clear boundary or hierarchy between the concurrent constructs and the sequential constructs in the specification diagram. Neither a vertical line nor a subdiagram necessarily represents a concurrent process.

Communicating Finite-State Machine

A Communicating Finite-State Machine (CFSM) [81] models multiple concurrent processes, which communicate with each other by passing messages through FIFO channels. Each concurrent process is described by one Finite-State Machine (FSM). FSMs are connected with directed channels to form a CFSM.

CFSM and MSG offer two dual views of a concurrent system: the former is a parallel composition of sequential machines, while the latter is a sequential composition of concurrent executions. Unlike actor specification diagram, CFSM has a clear hierarchy between concurrent constructs and sequential constructs. The sequential constructs are one level lower in the hierarchy than the concurrent constructs, as Each automaton in CFSM represents exactly one concurrent process. In addition, the temporal orders between events are not explicitly specified in CFSM.

Multiparty Session Types

Multiparty session types [86] are a type system for asynchronous communication sessions involving multiple peers. The theory is presented as typed calculus for concurrent processes. Recently, it also has been extended to the Actor model [87, 88]. A local session type describes the behaviors of one actor. Local session types can be composed into a global session type that describes the global behaviors of the actor system. On the other hand, a global session type can be projected onto an individual actor to derive a local session type. Similar to actor specification diagram, there are constructs for sequence, choice, recursion, parallelization, and message exchange in multiparty session types. However, multiparty session types do not have a graphical notation that clearly specifies the temporal orders between events.

4.2 OVERVIEW OF INFERENCE METHOD

We propose a method that combines static analysis and dynamic analysis to automatically infer the actor specification diagram from the system implementation. Figure 4.4 shows the overview of our inference method, which operates in two phases. In the first phase, we use static analysis to infer an abstract actor specification diagram from the system code. The

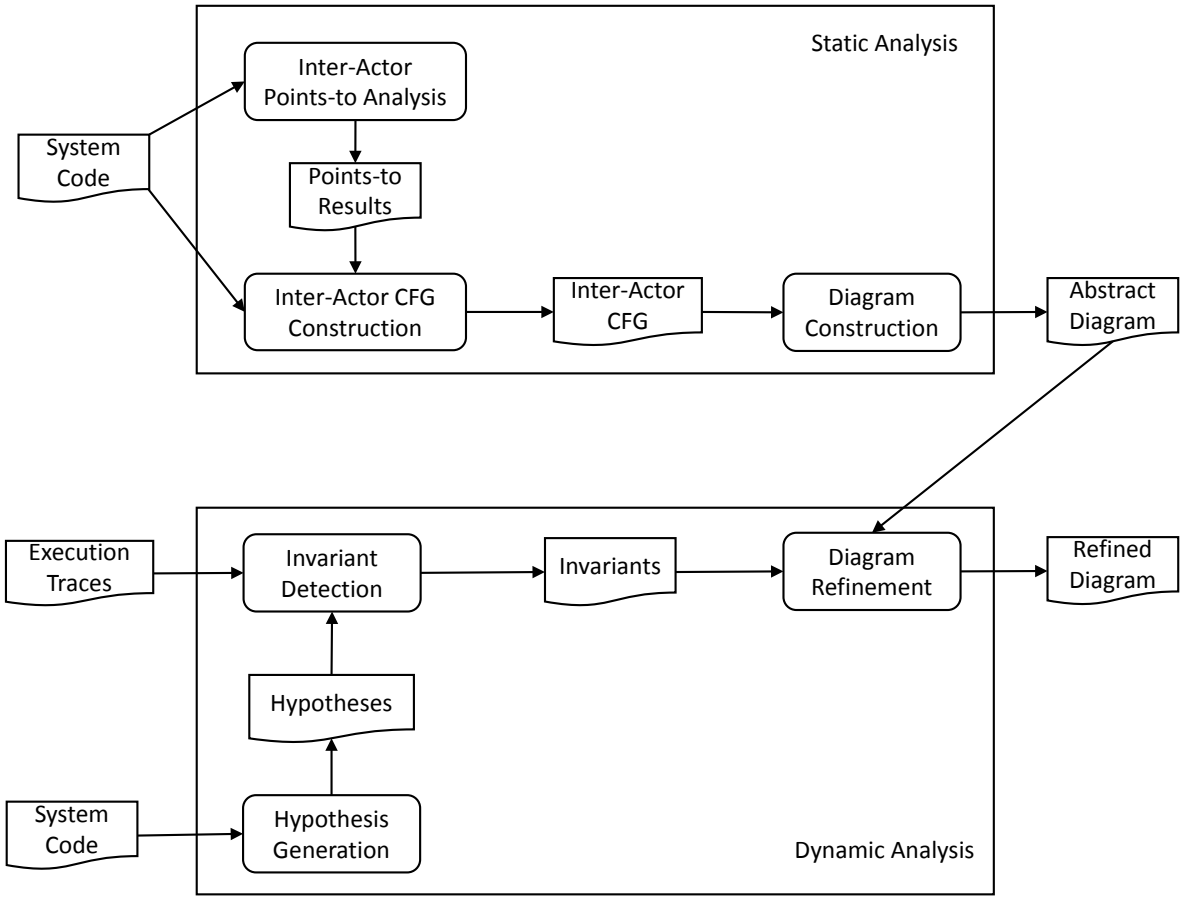


Figure 4.4: Overview of the inference method.

static analysis takes as input the system code, and outputs the abstract actor specification diagram in three steps. In the first step, it performs an inter-actor points-to analysis to compute the system-wide points-to information, which is used for resolving messages as well as message senders and recipients in later analysis. In the second step, it builds the control flow graph (CFG) for each individual actor and composes an *inter-actor control flow graph* (IA-CFG) by connecting the CFGs of individual actors. Two actors are connected if they exchange messages. The points-to analysis results from the first step are used to resolve senders, recipients, and message types in IA-CFG construction. The constructed IA-CFG describes the control flows within each actor as well as the message flows between actors. In the last step, an iterative algorithm is used to gradually construct the actor specification diagram based on the IA-CFG. After this step, we have constructed an abstract specification diagram, which soundly captures all possible actor operations and specifies the temporal

orders between these operations.

In the second phase, we use dynamic analysis to refine the abstract specification diagram. The dynamic analysis takes as input the system code and a set of execution traces, along with the abstract specification diagram from the first phase, and outputs a refined specification diagram. The analysis first infers dynamic invariants of the actor system based on the system code and execution traces. The inference process includes four steps: (1) identify variables in scope and generate hypotheses to be validated, in the form of relationships among variables in scope; (2) instrument the system code to collect values of the variable in scope at specified program points; (3) run the instrumented system with a set of inputs and generate a value trace for each input; (4) check each hypothesis against the set of value traces and output an invariant if the hypothesis holds on all value traces. The inferred dynamic invariants are then used for refining the abstract specification diagram generated in the first phase. The refinement targets at typical types of imprecisions introduced by static analysis, and improves the precision by removing false positives, instantiating loop iterations, splitting abstract actors and so on. The final output of the dynamic analysis is a refined diagram, which satisfies all the inferred invariants.

4.3 STATIC CONSTRUCTION OF SPECIFICATION DIAGRAM

In this section, we describe how we statically construct an abstract specification diagram from the system code. By *abstract* specification diagram, we mean a diagram which captures an over-approximation of the behavior of a system. The actors in the diagram are abstract actors (e.g., actor classes), which are merged from actor instances in concrete executions. In our analysis, the abstract specification diagram represents behaviors between actor classes. There are three core components in our static analysis: *inter-actor points-to analysis*, *inter-actor CFG construction*, and *specification diagram construction*. We next describe each of these components in detail.

4.3.1 Inter-Actor Points-To Analysis

To statically infer interactions between actors, an inter-actor points-to analysis is required to resolve the message, the sender and the recipient at each message-sending site. In contrast to an intra-actor analysis which isolates individual actors, inter-actor analysis is a system wide analysis across all actors. An inter-actor analysis connects actors by passing the points-to results computed within one actor to other actors through messages in message exchange or constructor arguments in actor creation. Existing points-to analyses for

sequential programs are not directly applicable for actor systems: such analyses treat actor library methods (e.g., the `tell` method for message send) as normal methods and analyze them without considering actor semantics. As a result, they often go deep into actor library methods and fail to establish connections between actors due to the complex multi-threading and networking code in these library methods. We adopt a state-of-the-art points-to analysis developed for actor systems in our previous work [89] to compute the system-wide points-to relation. The analysis provides a formal semantic model for actor operations: it specifies how each actor operation updates the points-to results globally. The semantic model of actor operations prevents the analysis from analyzing actor library code; this is the key to the scalability of the inter-actor analysis.

Our inter-actor points-to analysis is built upon the intra-actor points-to analysis of each actor class. The intra-actor point-to analysis is in fact the same as the inter-procedural points-to analysis for sequential programs, where the entry point is the message handler of the actor.

Recall that an MFG is a directed graph where: (1) each node represents an actor; (2) each edge represents that the sender actor either creates or sends a message to the recipient actor; and (3) the label of the edge represents the constructor arguments or the message sent. Our analysis is as follows:

1. We first build the initial segment of a *message flow graph* (MFG) [89], starting out from the receptionist actor.

   (a) We start with the receptionist actor, and perform the intra-actor points-to computation on this actor.

   (b) With the points-to results computed, we resolve (1) the actor created in each actor-creation site, and (2) the message sent as well as its recipient in each message-sending site.

2. We iteratively repeat the intra-actor points-to computation on the successor actors of the actor in the MFG: we add any created actors or message recipients as the successors of the receptionist actor to the MFG. Then we perform the same analysis on these successors.

3. This process repeats until a fixed point is reached when there are no updates on the MFG. This produces the system wide points-to results across all actors.

Algorithm 4.1 shows the iterative algorithm implemented with a worklist. We keep all actors to be analyzed in the worklist, and process one actor at a time in a FIFO order until

Algorithm 4.1: Inter-Actor Points-to Analysis

```
1  worklist ← [ ]
2  add all receptionist actors to worklist
3  while worklist not empty do
4      a ← worklist.removeFirst()
5      get the set of incoming edges E for a in MFG
6      if edgeMap[a] ≠ E then
7          edgeMap[a] ← E
8          IntraActorPointsToAnalysis (a, E, ptResult)
9          foreach actor creation in a do
10             resolve the actor being created a′
11             resolve the constructor arguments C
12             add the edge (a, a′, C) to MFG
13         foreach message send in a do
14             resolve the message recipient a′
15             resolve the message m
16             add the edge (a, a′, m) to MFG
17         add all successors of a in MFG to worklist
18 return ptResult
```

the worklist becomes empty. That is when the fixed point is reached. We start with an empty worklist and an empty MFG, and push all receptionist actors to the worklist (Line 2). The body of the while loop is the core of the algorithm. In each iteration, we retrieve the first actor in the worklist and check whether it is necessary to analyze the actor in this loop iteration (Line 6). To determine this, we maintain an `edgeMap` from actor nodes to sets of edges, which keeps track of the incoming edges for each actor node. We track this information because the set of incoming edges of an actor carries points-to information from its predecessors, and is an input to the analysis of this actor. We compare the current set of incoming edges of an actor computed from MFG (Line 5) to the corresponding set of incoming edges stored in the `edgeMap`. If they are the same, then we can safely skip the actor in this iteration, because the analysis input for this actor has not changed since last time. Analyzing this actor with the same input would produce the same result (i.e., neither the MFG nor the points-to results would be updated). Otherwise, we update the `edgeMap` (Line 7), and proceed to analyze this actor (Lines 8 - 16).

For the actor under analysis, we first perform an intra-actor points-to analysis (Line 8). The details of the points-to analysis are omitted in Algorithm 4.1 because it is a standard inter-procedural points-to analysis within the actor. We then identify all actor-creation and message-sending sites in the actor. For each actor-creation or message-sending site, we use

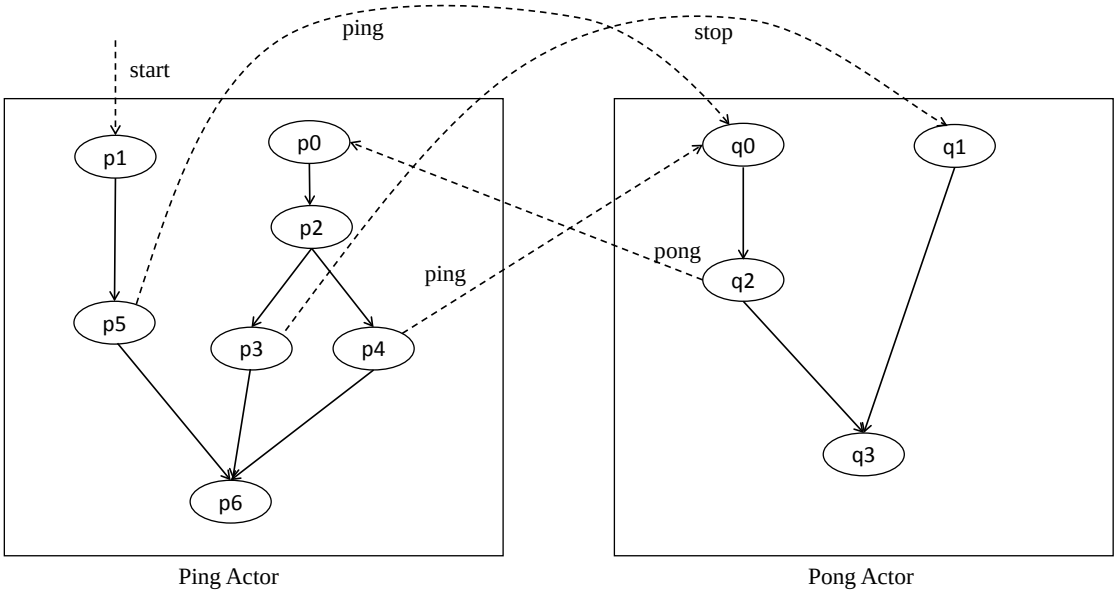


Figure 4.5: IA-CFG for the pingpong actors

the intra-actor points-to analysis results to resolve, respectively, the created actor with its constructor arguments, or the message recipient and the message send to it. We add the actor-creation or the message-sending edge to the MFG for each site. Now that the MFG is updated, we need to run this analysis again on the affected actors to propagate the change. The potentially affected actors are the successors of the actor under analysis. Thus, we append all the successors of this actor to the worklist for analysis (Line 17). The algorithm terminates when the worklist becomes empty, indicating that a fixed point is reached. We then return the inter-actor points-to results computed during the MFG construction. See Chapter 3 for a formal description of the inter-actor points-to analysis.

4.3.2 Inter-Actor Control Flow Graph Construction

With the results from the inter-actor points-to analysis, we construct the inter-actor control flow graph (IA-CFG) for the actor system. An IA-CFG is a directed graph, where each node is a CFG for an actor, and each edge represents a message flow from one actor to another, labeled with the message type. Note that IA-CFG is different from MFG: an IA-CFG captures not only the message flows between actors but also the control flows within each actor. However, because the specification diagram to be inferred does not have the construct for actor creation, an IA-CFGdoes not contain any actor-creation edge.

The first step for IA-CFG construction is to build the CFGs for each individual actor. This CFG is an inter-procedural CFG of the actor's class together with the message handler method as the entry point. We use standard techniques to construct the inter-procedural CFG by building an intra-procedural CFG for each method and connecting them via a call graph. In the next step, we compose these inter-procedural CFGs into an IA-CFG as follows. For each message-sending site, we resolve the message sent and the message recipient based on the inter-actor points-to results. We then add a directed edge, labeled with the message type, from the node containing this message-sending site to the entry point node of the inter-procedural CFG of the recipient actor. As a result, the inter-procedural CFGs are glued together as an IA-CFG whose entry points are the receptionists, because they are the only actors that can receive messages from outside.

An essential step towards the scalability of our analysis is to reduce the size of the intra-actor CFGs. Given that our goal is to infer the *restricted* actor specification diagrams that contain only message send-receive operations, we can safely remove instructions that are not message-sending site from the CFGs. In fact, we construct a *reduced CFG* containing only message-sending sites and control flow nodes rather than a full CFG.

Formally, a control flow graph G is a reduced graph of another control flow graph G' if G is a subgraph of G', and for every path p in G, there exists a path p' in G' so that p is a subsequence of p'. To construct the reduced CFG containing only message-sending sites for each actor, we first remove instructions that are not message-sending site from the full CFG. After this step, some CFG nodes may be empty (i.e., contain no instructions). We remove these empty CFG nodes except for entries, exits, branching nodes (nodes with multiple successors) and loop headers so that a reduced CFG containing only the message-sending sites is constructed. Finally, to facilitate our later analysis, we make the following changes to the reduced CFG: (1) split the entry point of the message handler method into multiple entry points so that each entry point accepts one type of messages; (2) split the CFG node containing multiple message-sending sites into multiple CFG nodes in sequence so that each split node contains exactly one message-sending site.

Figure 4.5 shows the IA-CFG for the pingpong actors. The solid arrows represent the control flows, and the dashed arrows represent the message flows. The subgraph in the left box represents the CFG of the Ping actor; the subgraph in the right box represents the CFG of the Pong actor. The two subgraphs are connected by the message flow edges. The CFG nodes p_3, p_4, p_5, and q_2 corresponds to the message-sending sites at Line 20, Line 17, Line 24, and Line 33 in Figure 4.2, respectively. Both actors have multiple split entry points, each of which accepts one type of messages. Nodes p_0, p_1, q_0, and q_1 accept *pong, start, ping, stop* messages, respectively. The *start* message is an external message, as the

message flow edge does not have a source node, indicating that the message is sent from the external environment. It is important to point out that in IA-CFG, loops can be formed across multiple actors by both control flow and message flow edges. In this example, nodes p_0, p_2, p_4 from the Ping actor, together with nodes q_0, q_2 from the Pong actor form a loop, of which q_0 is the entry and p_2 is the exit. This loop reflects the ping pong iteration between these two actors.

Although IA-CFG captures both control flows within individual actors and message flows between actors, it does not explicitly indicate the temporal orders between actor events because it lacks constructs for specifying temporal orders. For instance, from the IA-CFG of the pingpong example in Figure 4.5, it is not straightforward to see that the *stop* message is always sent after the *ping* and *pong* messages. Additional reasoning is needed in order to figure out the temporal orders between these actor events. Moreover, IA-CFG also does not have constructs for specifying the number of iterations for loops. Therefore, we convert IA-CFG to actor specification diagram, which is more expressive for specifying actor system properties.

4.3.3 Conversion To Specification Diagram

The idea of converting IA-CFG to actor specification diagram is to "execute" the IA-CFG so that temporal orders between events are captured during the execution and thus explicitly specified in the specification diagram. At the high level, the executor is triggered by an external message and gradually builds the specification diagram by converting the constructs in IA-CFG to their counterparts in specification diagram. In particular, a branch construct in IA-CFG is converted to a choice construct in a specification diagram; a message flow edge in IA-CFG is converted to message send-receive construct along with a parallel construct in a specification diagram. Abstractly, the resulting specification diagram can be viewed as the execution trace of the IA-CFG.

Algorithm 4.2: IA-CFG To Specification Diagram

Input : an inter-actor control flow graph `IA-CFG`
Output: a specification diagram D

1 initialize an empty diagram D
2 $E_0 \leftarrow$ `GetExternalMessage` (`IA-CFG`)
3 $L \leftarrow$ `PlaceReceiveEvent` (D, E_0)
4 $G_m \leftarrow$ `MessageHandlerCFG` (`IA-CFG`, E_0)
5 `ConstructDiagram` (`IA-CFG`, D, L, G_m)
6 **return** D

Algorithm 4.3: `ConstructDiagram`(`IA-CFG`, D, L, G)

```
1  worklist ← [ ]
2  n ← G.entry
3  while n ≠ G.exit do
4      E ← n.event
5      if E ≠ Null then
6          append E to L
7          E′ ← PairedReceiveEvent(E)
8          worklist.add(E′)
9      if n is a loop preheader then
10         G_l ← LoopBodyCFG(IA-CFG, n)
11         initialize a diagram D_l with a new vertical line L_l
12         ConstructDiagram (IA-CFG, D_l, L_l, G_l)
13         D_L ← Loop(D_l, ∗)
14         append D_L to L
15         n ← successor of loop exit
16     else if n is a branching node then
17         foreach branch b_i of n do
18             G_{b_i} ← BranchCFG(IA-CFG, n, b_i)
19             initialize a diagram D_{b_i} with a new vertical line L_{b_i}
20             ConstructDiagram (IA-CFG, D_{b_i}, L_{b_i}, G_{b_i})
21         D_C ← Choice(D_{b_1}, ... , D_{b_k})
22         append D_C to L
23         n ← join node of all branches
24     else if n has one normal successor then
25         n ← n.successor
26 while worklist not empty do
27     E_r ← worklist.removeFirst()
28     L_r ← PlaceReceiveEvent (D, E_r)
29     G_m ← MessageHandlerCFG (IA-CFG, E_r)
30     ConstructDiagram (IA-CFG, D, L_r, G_m)
```

Algorithm 4.2 shows the pseudo code for converting an IA-CFG to a specification diagram. The input to the algorithm is an inter-actor control flow graph, and the output of the algorithm is a specification diagram. The algorithm first initializes an empty specification diagram D (Line 1). It then invokes three procedures (Lines 2-4) to obtain: (1) the external triggering event E_0, (2) the vertical line L to place the event E_0, and (3) the CFG G_m to be executed for handling the event E_0. Finally, these three values, together with the `IA-CFG`, are passed as arguments to the procedure `ConstructDiagram`. `ConstructDiagram` uses this information to construct the specification diagram (Line 5).

We next describe the major procedures in the algorithm. Procedure `GetExternalMessage`

takes as input an IA-CFG and returns a receive event that represents the external message received at the entry point of the IA-CFG. Procedure `PlaceReceiveEvent` takes as input a diagram and a receive event, places the event on the diagram properly in terms of temporal orders, and returns the vertical line on which the event is placed. The vertical line can be either an existing line in the diagram or a new line created in the procedure. To keep the resulting specification diagram small, this procedure creates a new vertical line in the diagram only when the event cannot be placed on any of the existing vertical lines without violating the correctness of temporal orders between events. In other words, we attempt to place the event on existing vertical lines whenever it is possible. Procedure `MessageHandlerCFG` takes as input an IA-CFG and a message receive event, identifies the handler method of this message, and returns the inter-procedural CFG of the message handler extracted from the IA-CFG. The returned CFG is to be executed upon the receipt of the message.

Algorithm 4.3 describes the recursive procedure `ConstructDiagram`, which is the core of specification diagram construction. The input arguments of the procedure are an IA-CFG, a specification diagram D, a vertical line L in D, and a CFG G that is a subgraph of the IA-CFG. The main functionality of the procedure is to construct a subdiagram for the CFG G by executing it, and modify the specification diagram D in place by appending the subdiagram to the given vertical line L. During the execution of G, there might be messages further sent out to other actors and trigger the execution of more message handlers. Hence, the procedure is invoked recursively to gradually build the final specification diagram. The procedure uses a worklist to store the messages sent out during the execution of the given CFG. The first while loop in the procedure handles the execution of the CFG G. The execution starts from the entry point of G, and traverses the graph until reaching the exit of G. For each CFG node n, it takes out the message send event E (if not null) in n, appends E to the given vertical line L, and pushes the paired receive event E' of E to the worklist (Lines 4 - 8). The pending receive event E', which would trigger the execution of another message handler, is to be processed later in the procedure. After adding E to the diagram, our next step is to move the program counter along the CFG. There are three cases:

1. n is a loop pre-header (the immediate dominator of a loop header), indicating that n is followed by a loop;

2. n is a branching node with multiple successors; or,

3. n has only one normal successor.

To handle the loop case (Line 9 - 15), we do not unroll the loop because the number of loop iterations is unknown. Instead, we directly convert the loop to a subdiagram D_L with

a loop construct and append D_L to the given vertical line L. To construct D_L, we extract the CFG G_l of loop body via the procedure `LoopBodyCFG`, construct the subdiagram D_l for G_l by calling `ConstructDiagram` recursively, and apply a `Loop` operator on top of D_l. Since we cannot infer the number of loop iterations statically, we use the notation $*$ (Line 13) to indicate that the number of loop iterations can range from 0 to ∞. After adding D_L to D, we move the program counter to the successor of the loop exit. To handle the branch case (Line 16 - 25), we use the similar method. We build a subdiagram for each branch via calling `ConstructDiagram` recursively, apply a `Choice` operator on all subdiagrams of the branches to form a single diagram D_C, and append D_C to the specification diagram D. After that, we move the program counter to the join node of all branches. To handle the third case (Lines 24 - 25), we simply move the program counter to the only successor of n.

After the execution of G finishes, we process the pending message receive events in the worklist one by one in the second while loop (Lines 26 - 30). For each message receive event, we call `ConstructDiagram` recursively to construct a specification diagram for executing the handler of the message. When the second while loop terminates, we have added the specification diagram for G to its parent diagram D.

4.4 SPECIFICATION DIAGRAM REFINEMENT

To refine a statically constructed abstract specification diagram, we infer dynamic invariants of the actor system, and use them to improve the precision of the specification diagram. The dynamic analysis is complementary to the static analysis, as it is able to discover system properties that are difficult to infer by static analysis. In this section, we describe the dynamic invariant inference and pattern-based diagram refinement in detail.

4.4.1 Dynamic Invariant Inference

We follow the standard approach to dynamic invariant inference proposed in daikon [42]. The input to the inference algorithm is actor system code together with a set of inputs to the actor system. The output of the inference algorithm is a set of likely invariants of the actor system. The dynamic invariant inference process includes the following steps:

1. **Hypothesis Generation..** Define a set of hypotheses to be validated. A hypothesis is a potential invariant, in the form of a relationship among variables, to be checked at a specific *program point* (e.g., method entry, exit, loop header). The variables involved in the hypothesis are *variables in scope*, whose values are to be collected at the specified

program points.

2. **Instrumentation..** Instrument the program to collect values of the variables in scope at the specified program points.

3. **Trace Generation..** Run the program with a set of inputs and generate a set of value traces.

4. **Invariant Validation..** Check each hypothesis against the set of value traces. A dynamic invariant is discovered if a hypothesis holds on all traces.

We next describe the details of each step in dynamic invariant inference.

Hypothesis Generation

Hypothesis generation is the key to the discovery of useful invariants: the final set of inferred invariants are a subset of the generated hypotheses. To discover high quality invariants, it is important to identify the right set of variables in scope, which are the building blocks of the hypotheses. Traditionally, the variables in scope for a sequential program are the manifest variables declared in the program. However, in the context of an actor system, the manifest variables alone are not able to capture important information of the system such as the temporal orders between actor events. To collect such information, we further include in scope the derived variables that describe meta information of the system, such as the number of a certain type of actors created, and the number of a certain type of messages sent, the timestamp of an actor event. These derived variables are often not explicitly defined in program, but their values can be derived from program states.

In particular, we define the following set of variables in scope,

- Configuration variables
- Method parameters
- Actor creation events
- Message send or receive events
- Derived variables
  - Number of created actors
  - Number of created actors of a specific type

- Number of messages sent by an actor
- Number of messages of a specific type sent by an actor
- Number of messages received by an actor
- Number of messages of a specific type received by an actor
- Number of actors created at a specific program location
- Number of messages sent at a specific program location
- Number of distinct senders at a specific program location
- Number of distinct recipients at a specific program location

A *message event* is uniquely defined as a tuple (T, s, r, M, l, t), where

- $T \in \{!, ?\}$ represents either a send or receive operation,
- s represents the sender,
- r represents the recipient,
- M represents the message content,
- l represent the program location of the event, and
- t represents the timestamp when the event occurs.

An *actor creation* event is uniquely defined as a tuple (a, l), where a represents the actor instance created, and l represents the program location of the event. The actor creation event does not contain a timestamp because the specification diagram does not include actor creation operations. The event is only used to record the actors created in the system. Once the values of the two types of events are collected, the values of all the other derived variables can be computed based on these events. Note that we do not include variables of local actor states in the invariant detection because the restricted specification diagram to be inferred does not specify local states. Excluding these variables significantly reduces the runtime of the invariant detection.

After identifying the set of the variables in scope, the hypotheses are generated over these variables in the form of the following two types of relationships:

- **Linear Relationship.** Linear relationships on one or two variables in scope, respectively defined as follows,

$$x = c \tag{4.1}$$

$$y = ax + b \tag{4.2}$$

where x, y are variables in scope, and a, b, c are constants. Note that we consider only integer constants in our dynamic invariant detection.

- **Temporal Relationship.** Temporal order between two message events is defined as a happens-before relation. A message event e_1 happens before a message event e_2, denoted as $e_1 < e_2$ if e_1 is *always* followed by e_2 in any execution trace, that is $t_1 < t_2$, where t_1, t_2 are the timestamps of e_1, e_2, respectively.

One optimization for reducing the analysis runtime is to filter out hypotheses that have already been entailed by the abstract specification diagram. For instance, if the abstract specification diagram has already identified the causal relationship, which indicates the temporal order between two events, then it is not necessary to generate hypotheses on the temporal order between these two events.

Instrumentation

We scan through the system code, identify variables in scope, and insert log statements to record values of these variables. To record message send events, we insert a log statement after each message-sending site, recording the id of the sender actor, the id of the recipient actor, message, and current timestamp. To record message receive events, we insert a log statement after the entry of each message handler method, recording the id of the sender actor, the id of the recipient actor, message, and current timestamp. To record actor creation events, we insert a log statement after each actor-creation site, recording the id of the created actor. Note that we obtain the location of each event statically rather than record them at the run time.

Trace Generation

To generate the value traces, we run the set of system inputs on the instrumented program, and extract the values of variables in scope from the system logs. Each system input generates one value trace. Based on the value trace, we further compute the values of the derived variables for each run. These computed values of the derived variables are used in invariant validation. It is important to note that we run all actors on a single machine so that they use a shared clock, and thus the timestamps of events on different actors are still comparable.

Invariant Validation

The last step is to validate each of the generated hypothesis against the set of value traces, and produce the invariants. Since our goal is to detect global properties of the actor system, we are interested to see what has happened in the system. Therefore, all of our hypotheses are validated at the *exit* of the actor system. To obtain the value of a variable at the exit of the actor system, we find the last occurrence of its value from the value trace. A hypothesis holds on a value trace if it is valid by plugging in the values of the variables at the exit of the system from this value trace. We report the hypothesis as an invariant only if it holds on all value traces.

4.4.2 Pattern-Based Refinement

With the inferred dynamic invariants, we further refine the abstract specification diagram. Static analysis can often result in imprecisions such as false positives due to its conservative nature. Our idea is to leverage the dynamic information to improve the precision of the statically inferred specification diagram, because certain system properties are very difficult to infer by static analysis, but can be discovered effectively by dynamic analysis. In particular, we use a pattern-based approach to refine the diagram, because we observe that there are common patterns of imprecision from static analysis. For each pattern of imprecision, we propose an adhoc refinement based on the dynamic invariants to eliminate the imprecision.

The specification diagram refinement is an iterative process. The first step is to match the specification diagram, along with the dynamic invariants against a set of predefined patterns. If a match is found, the corresponding refinement for this pattern is applied to the specification diagram to produce a refined specification diagram. Then we validate the refined specification diagram against all the dynamic invariants. If all dynamic invariants hold on the refined specification diagram, the refinement succeeds. Otherwise, we roll back to the specification diagram before the refinement. This iterative process continues until there is no applicable refinement.

Compared to the generic refinement approaches such as CEGAR [82], the advantage of the pattern-based approach is that it is able to quickly identify the imprecisions and eliminate them in an effective way. However, the disadvantage of the pattern-based approach is that it lacks generality, and may not work for all diagrams. For instance, it may miss some refinement opportunities if none of the predefined patterns is matched. We choose the pattern-based approach because we have observed that there are only a few common patterns in most of imprecisions caused by typical limitations of static analysis. We next describe

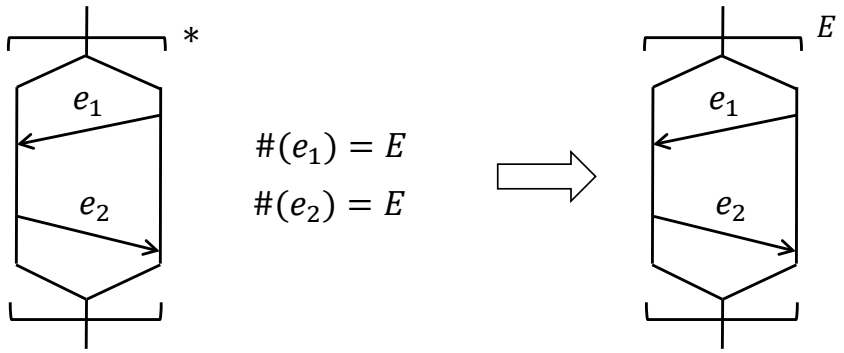


Figure 4.6: Instantiate the number of loop iterations.

these common patterns of imprecision and their corresponding refinement.

Instantiate Number Of Loop Iterations

Although loops can be identified statically in IA-CFG, it is often difficult for static analysis to figure out the exact number of loop iterations. Constant propagation techniques might be used to derive the number of loop iterations when it is a constant. However, in many cases, the number of loop iterations is a variable, not a constant. In such cases, constant propagation techniques would not work, and we have to conservatively assume that the loop iterations can be any number. This is a typical limitation of static analysis, which results in imprecisions in the specification diagram.

Figure 4.6 shows this pattern of imprecision and the corresponding refinement. On the left hand side, the specification diagram is imprecise, as it contains the notation $*$, which can range from 0 to ∞. The dynamic invariant specifies that both the number of event e_1 and the number of event e_2 in the loop are always E, where E can be a constant, a variable in scope (e.g., configuration variable), or expressions formed by variables in scope. Based on this invariant, we infer that the number of loop iterations is E, and then refine the specification diagram on the left hand side to a more precise one on the right hand side.

Remove False Positives

Static analysis often introduces false positives — statically constructed specification diagrams may include behaviors that can never happen in concrete executions. Figure 4.7 shows a pattern of imprecision, where the specification diagram contains a false positive branch

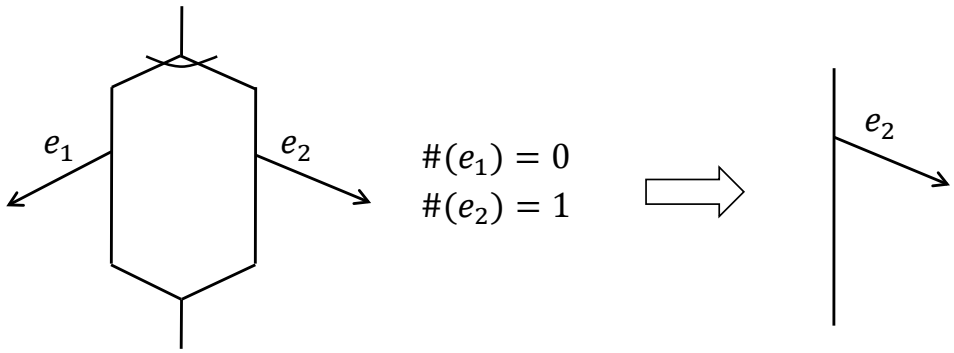


Figure 4.7: Remove false positive branch.

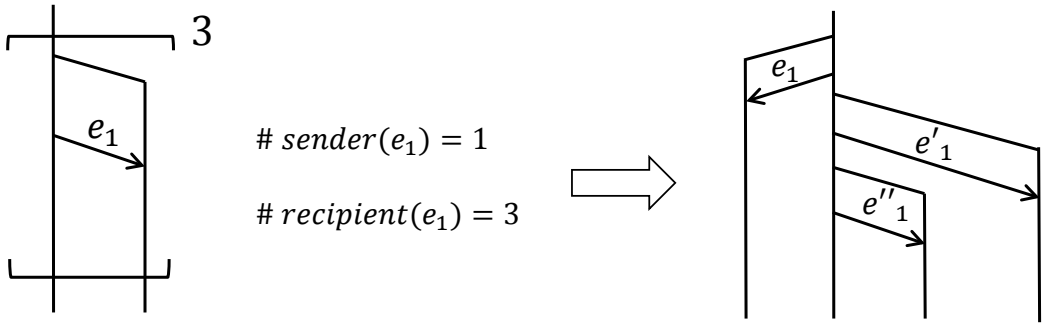


Figure 4.8: Split abstract actors.

(i.e., a branch that can never be taken). In this case, the specification diagram contains a choice operator, indicating that either event e_1 or event e_2 would happen. However, the dynamic invariant specifies that the number of event e_1 is always 0 and the number of event e_2 is always 1. In other words, event e_2 always happens while event e_1 never happens. Based on this invariant, the refinement simply removes the branch of event e_1.

This pattern of imprecision is quite typical in the inferred abstract specification diagrams. For example, the points-to analysis often contains false positives, which eventually lead to false positives in the specification diagram. When we resolve the recipient in a message-sending site, the points-to analysis concludes that the recipient can be either actor a_1 or actor a_2. As a result, the specification diagram contains a choice operator, specifying that this message can be sent to either of these two actors. However, in all concrete executions, the recipient can only be actor a_1 at this message-sending site. Therefore, the other branch for actor a_2 is a false positive, and should be removed.

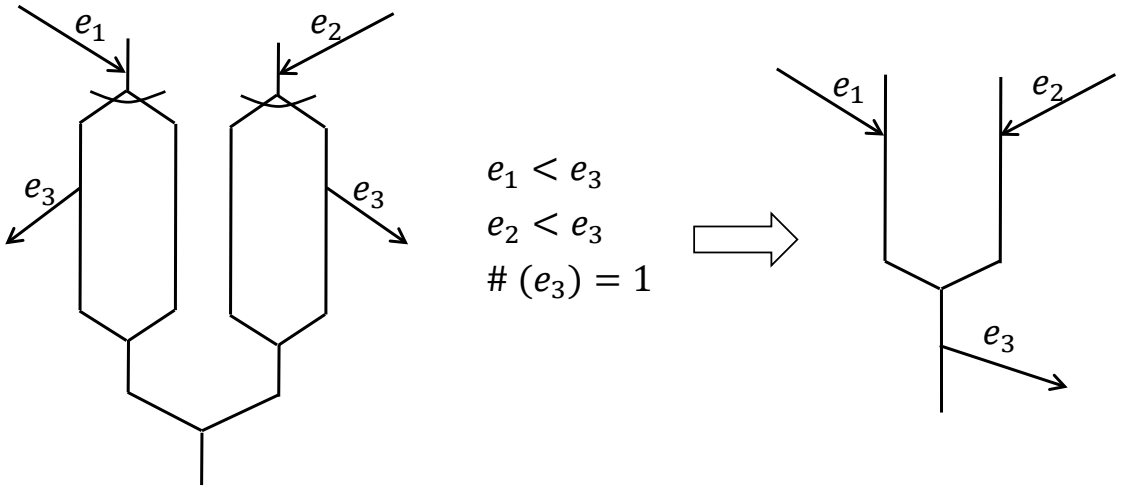


Figure 4.9: Infer additional temporal order.

Split Abstract Actors

Recall that our static analysis creates one abstract actor for each actor class. This strategy results in imprecisions in the specification diagram, as multiple concrete actor instances of the same class are merged into a single abstract actor. The abstract actor includes the behaviors of all its corresponding concrete actors. To eliminate such imprecision, we need to split the abstract actors into concrete actors.

A typical refinement for this pattern of imprecision is to split an abstract actor into multiple behaviorally identical concrete actors. Figure 4.8 shows such an example, where an actor of class A sends a message of class M to three distinct recipient actors of class B. This behavior is implemented in code as a loop of three iterations. In each loop iteration, the same sender actor sends a message to a distinct recipient actor. The abstract specification diagram for this implementation is shown on the left hand side in the figure. It contains only two abstract actors: one for class A and one for class B. However, the dynamic invariants indicate that there is always one sender of class A and that there are three distinct recipients of class B from this message-sending site. These three recipient behave the same. Therefore, the refinement is to split the abstract actor of class B into three behaviorally identical actors, and the sender actor sends a message of class M to each of them.

Add Temporal Order

Although static analysis is capable of inferring temporal orders between events enforced by control flows and message flows, it often fails to infer another type of temporal orders

enforced by coordination constraints. Coordination is an essential part of actor systems so that actors can work together to accomplish common tasks. While coordination happens, there are temporal orders between actor events being enforced. Such temporal orders are critical to be captured in the specification diagram in order to understand the actor system. Since there is no coordination construct in the Actor model, coordination in actor systems are often implemented by placing conditions on actor local states. Such practice makes it difficult for our static analysis to infer the coordination constraints and thus the temporal orders enforced by them, because our static analysis does not analyze on actor local states. However, dynamic analysis can effectively capture these temporal orders, because they are naturally reflected in concrete executions. Therefore, we can refine the abstract specification diagram by adding these detected temporal orders to reflect the coordinations across actors.

Figure 4.9 shows an example of refining the specification diagram by adding additional temporal orders detected as dynamic invariants. Two messages in events e_1 and e_2 are sent to an actor a. Actor a sends a message out in event e_3 only when it has received both messages in e_1 and e_2. Otherwise, a it does nothing on receiving a message. The specification diagram on the left is produced from our static analysis. It conservatively assumes that upon receiving a message, a either does nothing or sends a message in e_3. However, the dynamic invariants indicate that event e_3 always happen after both events e_1 and e_2, and e_3 happens only once. Thus, the refinement is to adjust the specification diagram to accommodate the additional temporal orders $e_1 < e_3$ and $e_2 < e_3$. In addition, the false positive branch introduced by static analysis is also removed in the refinement, because event e_3 always happens and happens exactly once.

4.5 IMPLEMENTATION AND EVALUATION

In this section, we first describe the prototype implementation of our inference method, and then present the evaluation of the method, including evaluation setup and evaluation results with case studies.

4.5.1 Implementation

We implement our inference method in a prototype tool named ASpec for "Actor Specification". ASpec is implemented in Java and works for programs written in the Java Akka framework. ASpec consists of three major components: (1) a static analyzer (∼2k LoC), (2) a dynamic invariant detector (∼3k LoC), and (3) a pattern-based refiner (∼1k LoC). The static analyzer constructs the abstract specification diagram, and is developed on top

of our previous tool Tap [89]. It re-uses the inter-actor points-to analysis component and the MFG construction component in Tap, and extends it by adding the functionalities of IA-CFG construction and specification diagram conversion from IA-CFG. The analyzer leverages Wala [66] as the underlying static analysis framework and uses its built-in analyses such as points-to analysis, control flow graph construction, and call graph construction.

Our dynamic invariant detector is implemented from the scratch, because the existing tools for sequential programs such as daikon [42] cannot be directly applied to actor systems. The invariant detector is developed with a focus on finding invariants on actor events and the temporal orders between them. It contains an instrumenter that inserts log statements at the source code level to record the values of variables in scope as well as a invariant checker to validate the hypotheses against value traces. The pattern-based refiner matches the specification diagram and the dynamic invariants against a list of patterns, and applies the corresponding refinement when a match is found. We have implemented the refinement for 6 patterns in the refiner including all the patterns described in the previous section. The refiner is designed in a generic way so that future patterns can be easily added to it.

4.5.2 Experiment Setup

We evaluate ASpec on a set of third-party *Savina* benchmarks as well as two real-world protocols Transmission Control Protocol (TCP) [90] and Session Initiation Protocol (SIP) [91]. The *Savina* benchmarks consist of 30 diverse programs written purely using actors. *Savina* has three categories: micro benchmarks with 8 well-known actor examples, concurrency benchmarks with 8 classic concurrency problems, and parallel benchmarks with 14 realistic parallel applications. See [41] for a more detailed description of the benchmarks. *Savina* has been used in the actor community for various evaluation purposes, such as performance comparison of actor languages/frameworks [41, 67], actor profiling [68], and mapping from message passing concurrency to threads [69]. The original *Savina* does not have a Java *Akka* implementation. We transformed the Scala *Akka* implementation in *Savina* into Java *Akka* and used the transformed version in our experiment because ASpec currently supports only Java *Akka*. We had at least two actor programmers double check that the transformed Java version is equivalent to the Scala version.

TCP is a communication protocol for reliable data transmission between hosts. It establishes a bi-directional connection between two hosts for data transmission through a three-way opening handshake. It terminates the connection through a four-way closing handshake. SIP is a signaling protocol for creating, modifying, and terminating VoIP calls and multimedia sessions with one or multiple participants. We did not find open source

actor implementations for these two protocols; we implemented a much simplified actor version of each protocol, which captures the essence of the protocol. The two protocols become complicated with many corner cases in the presence of packet losses and packet delays. Since our goal was to model the common-case behaviors of the protocols, we did not explore these corner cases. This is similar to what has been done in previous research [75]. In our actor implementations, we assumed exactly-once FIFO message delivery of the underlying infrastructure. Our implementations also omitted numerous details of the protocols such as TCP header format, but focused on message exchanges between participants.

To produce enough execution traces for dynamic invariant inference, we not only included all existing tests, but also randomly generated up to 100 tests for each benchmark. A test consists of message contents and deterministic schedules of message receive events on each actor. We implemented a tool for random test generation, consisting of a data generator and a schedule generator. The data generator randomly picks values for a give type. The schedule generator intercepts the mailbox of each actor, and shuffles the messages in the mailbox.

To evaluate the accuracy of our inference method, we manually derived one diagram as the reference diagram for each subject (in total 32 reference diagrams), and compared the inferred diagram on each subject to the corresponding reference diagram. An inferred diagram is accurate if it is behaviorally equivalent to the reference diagram. Again, we manually checked the behavioral equivalence between two diagrams. The main threat to validity of our evaluation is that there might be human errors in deriving the reference diagrams and checking the equivalence between the reference diagram and the inferred diagram. To mitigate this threat, we had at least two people double check the correctness of the reference diagrams and the equivalence checking.

4.5.3 Evaluation Results

Overall, ASpec is able to infer succinct and accurate specification diagrams for actor systems. It successfully infers the correct specification diagrams for 25 out of 32 (78%) subjects: 23 out of 30 benchmarks and 2 out of 2 real-world protocols. The evaluation results are summarized as follows.

- ASpec is capable of inferring specification diagrams that captures all possible behaviors of actor systems. It is also highly reliable in inferring temporal orders that are enforced by control flows and causal relationships. Thanks to the soundness of our static analysis, all system behaviors and temporal orders in each subject are included

in the inferred abstract specification diagram.

- ASpec is able to infer coordination constraints that involve multiple actors. A common example of such coordination constraints is that a message can only be sent after the actor receives certain messages from multiple other actors. Some coordination constraints are inferred statically from the control flow structures. Others are inferred from dynamic invariants.

- The patten-based refinement is effective in improving the precisions of the abstract specification diagrams. It successfully refines 22 imprecise abstract specification diagrams to the accurate diagrams. The refinement for common patterns includes removing false positive branches, splitting abstract actors, and finding additional temporal orders etc. The pattern-based refinement is very efficient – it typically finishes within 10 iterations.

- ASpec is able to identify implicit loops formed across multiple actors by both control flow edges and message flow edges through loop detection on IA-CFG.

- ASpec is accurate in inferring the number of loop iterations using dynamic invariants. It correctly instantiates the number of iterations for 15 out of 17 loops.

There are seven subjects on which ASpec fails to infer the accurate specification diagram. We investigate these seven cases and find the following major issues causing the failures. Note that some of the issues overlap on subjects (i.e., one subject has multiple issues).

- On three subjects, ASpec fails to infer the coordination constraints, which are too complex to infer by static analysis, and are beyond the scope of our targeted dynamic invariants (e.g., non-linear invariants).

- On two subjects, the pattern-based refinement fails to split the abstract actors because the concrete actors are not behaviorally equivalent. In such cases, it is difficult to precisely split an abstract actors into multiple behaviorally different actors.

- On one subject, ASpec fails to infer the number of iterations for two loops, because they are in a quadratic relationship (non-linear relationship) with some variables in scope.

- On two subjects, the refinement does not work, because the specification diagrams and the dynamic invariants do not match any of the patterns.

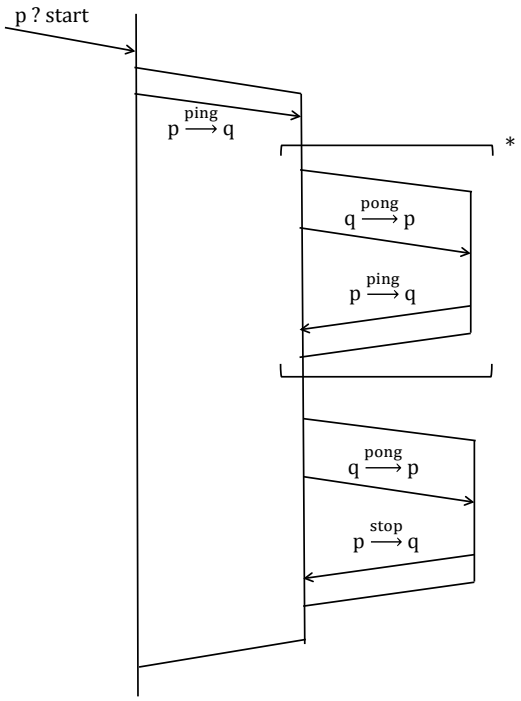


Figure 4.10: Abstract specification diagram of pingpong actors.

The above issues can be addressed by developing more advanced techniques to infer the complex coordination constraints, combining generic refinement techniques such as CEGAR with our pattern-based refinement, and expanding the forms of the hypotheses for dynamic invariants to non-linear relationships. We will leave these to the future work.

We next present four case studies from our evaluation, namely the pingpong benchmark, the master-worker benchmark, the dining philosophers benchmark, and the TCP protocol.

Pingpong

Although the pingpong benchmark is one of the smaller subjects, it contains a number of interesting findings and demonstrates the effectiveness of ASpec. There are three obstacles in the path of reaching the accurate specification diagram: (1) identify the implicit loop formed by the ping pong messages between the ping actor and the pong actor; (2) discover the coordination constraint that the ping pong loop begins after the `start` message is received, and ends before the `stop` message is sent; (3) discover the number of iterations for the ping pong loop. Inferring these specifications automatically and accurately is a non-trivial task.

ASpec successfully infers the accurate specification diagram, demonstrating the power of combining static analysis and dynamic analysis. The code of the pingpong benchmark

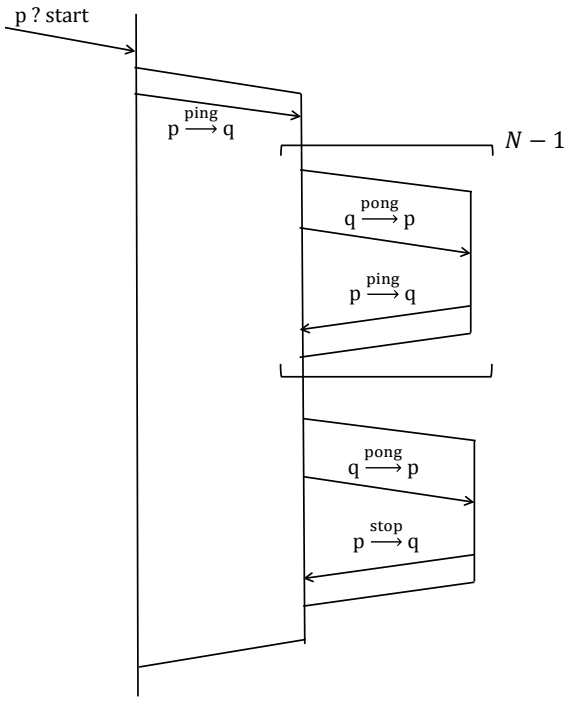


Figure 4.11: Refined specification diagram of pingpong actors.

is shown in Figure 4.2 and its correct specification diagram derived manually is shown in Figure 4.3. The first intermediate product from ASpec is the IA-CFG shown in Figure 4.5. From the IA-CFG, ASpec is able to detect the ping pong loop formed by the nodes p_0, p_2, p_4 from the ping actor, and the nodes q_0, q_2 from the pong actor. The loop body in the inferred specification diagram contains exactly the ping pong message flows. Note that the loop is implicit in that there is no corresponding loop statement in the code. It cannot be detected without a global model of the system (IA-CFG in this case).

Furthermore, the structure of the loop implies the coordination constraint. The entry of the loop is q_0, which can only be reached through the path $p_1 p_5 q_0$. Due to the causal relationship, we can conclude that the events in loop body must happen after the receipt of the `start` message. Similarly, the only exit of the loop is p_2, which is the immediate dominator of p_3, and the edge (p_3, q_1) represents the send-receive of the `stop` message. Therefore, the `stop` message is sent after all events in the loop body. Figure 4.10 shows the abstract specification diagram converted from the IA-CFG. With only static analysis, we are able to identify the implicit ping pong loop and infer the coordination constraint.

However, the abstract specification diagram is still imprecise, as the number of the ping pong iterations is unknown. ASpec accurately refines the abstract specification diagram in Figure 4.10 to the final specification diagram shown in Figure 4.11 by instantiating the $*$

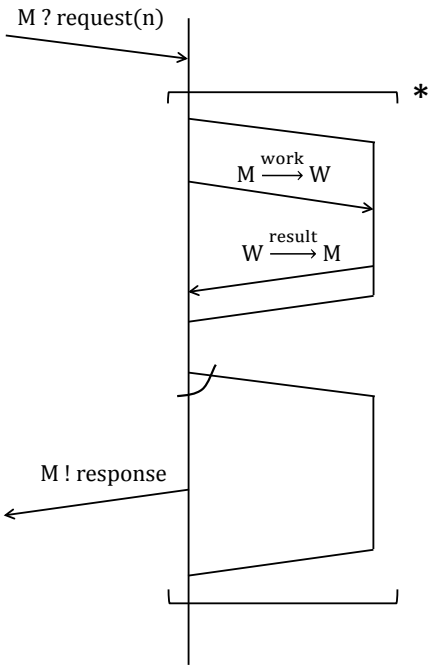


Figure 4.12: Abstract specification diagram of master-worker actors.

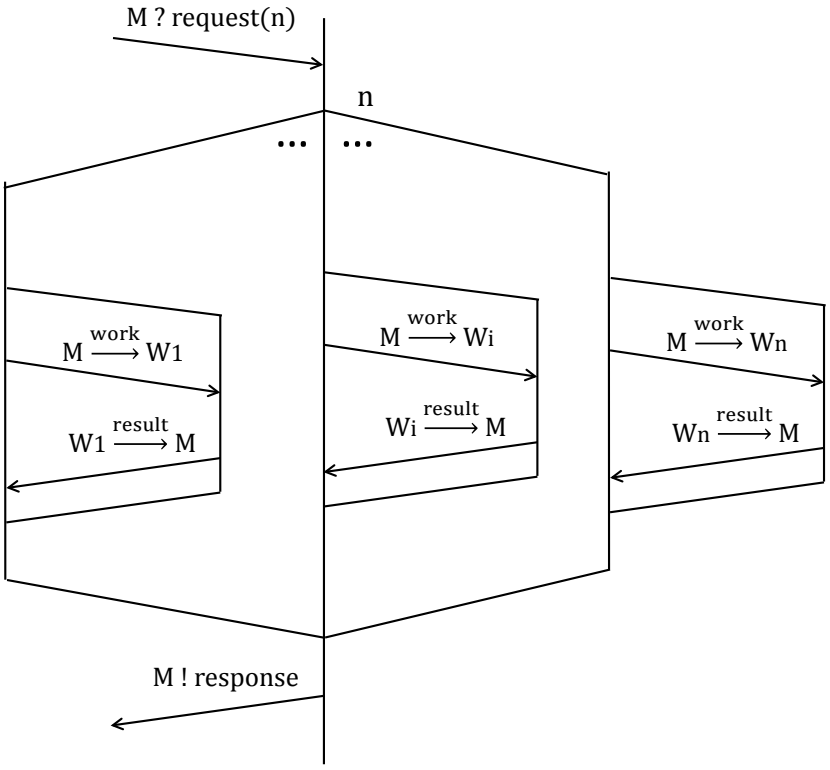


Figure 4.13: Refined specification diagram of master-worker actors.

notation with $N-1$. The inference of the number of loop iterations is based on the dynamic invariants that there are in total N ping messages sent and N pong messages sent in the system. In the abstract diagram, since there is exactly one ping message and one pong message sent outside the loop, there must be $N-1$ ping messages and $N-1$ pong messages sent inside the loop. Hence, we conclude that the number of loop iterations is $N-1$. Note that although the final specification diagram inferred by ASpec in Figure 4.11 looks different from the manually derived one in Figure 4.3, they are behaviorally equivalent.

Master Worker

The master-worker benchmark is a typical parallelism application. A single master receives requests from clients, distributes the workloads to a bunch of workers to process in parallel, and responds to the clients after aggregating the processed result from each worker. To produce the accurate specification diagram, we need to infer the number of workers in the system as well as discover the coordination constraint that the response is sent after the master receives the result from each worker.

Figure 4.12 shows the abstract specification diagram constructed by ASpec. We use `M` and `W` to represent the master and the worker actors respectively. The specification diagram captures all possible message exchange events in the system, including the receive of the `request` message, the send-receive of the `work` message and the `result` message, and the send of the `response` message. However, the abstract specification diagram is imprecise – there is only one abstract worker actor in the diagram, as multiple concrete worker actors are merged into the abstract actor. The loop in the diagram corresponds to a loop in code, where the master sends the `work` message to each worker. From the static analysis point of view, since the concrete workers are merged into one, the analysis sees the interactions between the master and the worker being repeated in a loop. Moreover, there are false positives in the abstract specification diagram. The abstract diagram specifies that whenever the master receives a `result` message from the worker, it either sends a `response` message out or does nothing. This is inferred from an if statement in code that checks whether the master has received the results from all workers. The master only sends out the `response` message once after receiving all `result` messages. Static analysis cannot figure out this coordination constraint, and thus has to conservatively assume that both branches can be taken upon the receipt of each `result` message.

Figure 4.13 shows the refined specification diagram by ASpec, which describes the system behaviors precisely. The refinement is enabled by some key dynamic invariants inferred by ASpec. ASpec discovers that the only instruction, in which a master sends a `work` message to

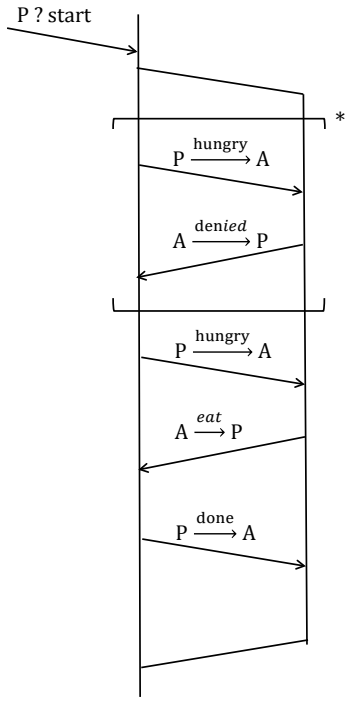


Figure 4.14: Abstract specification diagram of dining philosophers.

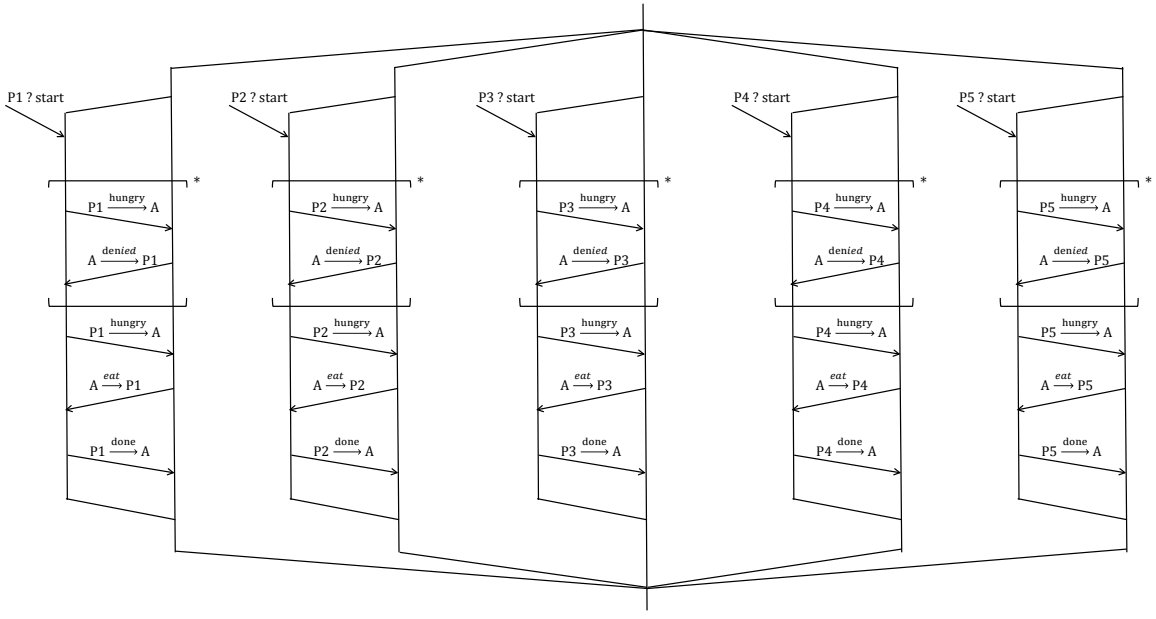


Figure 4.15: Refined specification diagram of dining philosophers.

a worker, is always executed n times, and n is the content of the `request` message received by the master. In each execution of this instruction, the master is the same actor, but the worker is distinct. This indicates that one master is sending a `work` message to n distinct workers. In addition, all the workers behave the same. Hence, we split the abstract worker into n behaviorally identical workers and eliminate the loop construct in the abstract specification diagram. ASpec also detects that the `response` message is always sent after all `result` messages, and the `response` message is sent exactly once. Based on these invariants, ASpec removes the branch construct following each receive of the `result` message, and appends one single send of the `response` message to the end of the diagram.

Dining Philosophers

The dining philosophers benchmark implements a classic concurrency problem. Five philosophers sit around a dining table to eat, with only five chopsticks on the table. One chopstick on the left and one on the right of each philosopher. A philosopher can eat only when he or she possesses the chopsticks on both sides. The benchmark is implemented with two actor classes: a philosopher class and an arbitrator class that determines whether a philosopher can eat based on the aforementioned rule.

Figure 4.14 shows the abstract specification diagram for the dining philosophers benchmark. We use `P` and `A` to represent the philosopher and the arbitrator actors respectively. The specification diagram precisely captures all the events between the philosopher and the arbitrator as well as the temporal orders between events. The philosopher first receives a `start` message and then sends out a `hungry` message to the arbitrator. If denied, the philosopher keeps sending the `hungry` message until an `eat` message is received from the arbitrator. Then the philosopher starts to eat. Once it is done, the philosopher sends a `done` message to the arbitrator to release the chopsticks. Again, our static analysis successfully infers the implicit loop between the philosopher and the arbitrator as well as the coordination constraint that the philosopher can eat only when he or she acquires both chopsticks from the arbitrator. However, the abstract specification diagram is still imprecise, as it merges the five philosophers into an abstract philosopher.

Figure 4.15 shows the refined specification diagram by ASpec. Similar to the master-worker benchmark, ASpec uses the same refinement to split the abstract philosopher into five behaviorally identical concrete philosophers, based on the detected dynamic invariants. It is interesting to point out that ASpec does not instantiate the number of iterations for the implicit loops formed by the `hungry` message and the `denied` message. This is correct because there is truly no invariant on the number of iterations for these loops.

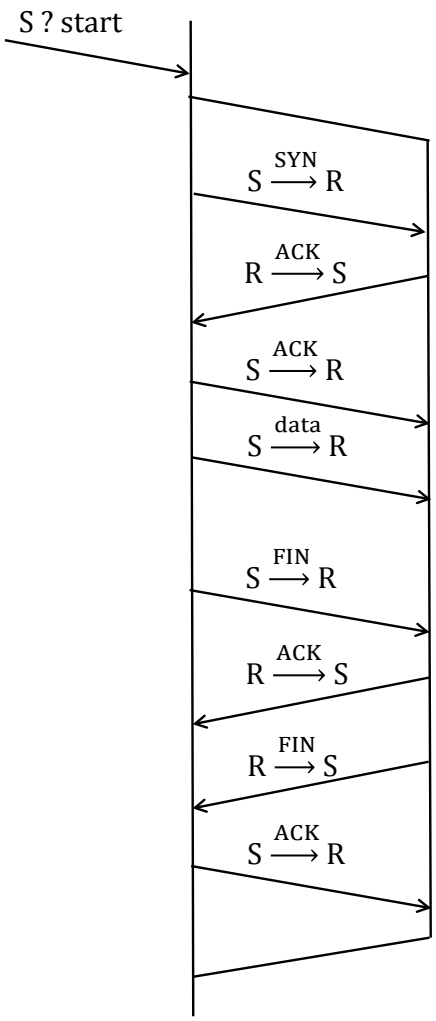


Figure 4.16: Specification diagram of TCP.

TCP

Our implementation of the TCP protocol simulates the normal use cases with one sender host and one receiver host. The sender establishes a connection with the receiver, and sends data to the receiver. After the data transmission is done, the sender closes the connection.

Figure 4.16 shows the specification diagram for TCP inferred by ASpec. We use `S` and `R` to represent the sender and the receiver actors respectively. The specification diagram produced by static analysis is already precise in this case. ASpec correctly infers the three-way opening handshake between the sender and the receiver for establishing the connection for data transmission. As shown in the diagram, the sender first sends `SYN` message to the receiver, the receiver responds with an `ACK` message. The sender further sends an `ACK`

message to the receiver, then the connection is established, and the sender starts to send `data` to the receiver. ASpec also correctly infers the four-way closing handshake between the sender and the receiver for closing the connection. The sender closes the connection on its end and sends a `FIN` message to the receiver to close the connection on the receiver's end. The receiver first sends back an `ACK` message, acknowledging that it has received sender's `FIN` message, and then sends back a `FIN` message of its own to notify the sender that it has closed the connection on its end. After receiving the `FIN` message, the sender sends an `ACK` message to the receiver. The TCP subject demonstrates that our static analysis is able to reliably infer the causal orders between actor events.

CHAPTER 5: RELATED WORK

In this chapter, we discuss related work on the directions of concurrency bugs studies, testing and static analysis on concurrent systems, and behavioral model inference for concurrent systems.

5.1 STUDIES ON RELIABILITY OF CONCURRENT SYSTEMS

5.1.1 Bug Characteristics Studies.

A lot of work has been done to study bug characteristics in software systems. Most of these studies share the same goal of improving reliability of software systems. Earlier work mainly studied bugs in large open-source software systems such as operating systems, databases. Chou *et al.* [92] studied bugs in Linux and OpenBSD that were detected by their static-analysis tool. Gu *et al.* [93] studied Linux kernel behaviors under injected bugs. Chandra *et al.* [94] studied bugs in open-source software from a failure recovery perspective.

Recently, there have been studies on concurrency bugs. Lu *et al.* [95] studied 105 randomly selected real-world concurrency bugs from 4 representative open source applications. Some of their interesting findings include that most concurrency bugs can be reliably triggered by less than 4 memory accesses, and the fixes of many concurrency bugs were not correct at the first try, indicating the difficulty of reasoning concurrent execution by programmers. Their study mainly focuses on programs using the multi-thread concurrency model, which is different from the concurrency model in our study.

Yuan *et al.* [96] studied 198 randomly selected, user-reported failures in distributed data-intensive systems. Their main finding is that the majority of catastrophic failures could easily have been prevented by performing simple testing on error handling code.

Asadollah *et al.* [97] studied 221 reports of concurrency bugs from open source Hadoop projects in the last decades. Their findings include that fixing time for concurrency and non-concurrency bugs is different but this difference is not big, and concurrency bugs are considered to be slightly more severe than non-concurrency bugs. Their study focuses on comparing concurrency bugs to non-concurrency bugs in terms of their fixing time and severity.

Kavulya *et al.* [33] studied failures in MapReduce programs. Although we shared some similar findings on failure workloads, there were essential differences between their work and ours. First, their subjects were jobs created by university research groups while ours

came from production jobs. Second, the SCOPE jobs in our study were written with a hybrid programming model whereas their Hadoop jobs were based on the MapReduce model. Different programming models may lead to different failure characteristics. Third, their work mainly focused on studying the workloads of running jobs for achieving better system performance such as job scheduling whereas our work focused on studying source code and input data of failed jobs for the purpose of failure reduction and fixing. There was work [98] that studied fault tolerance in MapReduce. However, the failures in their study referred to failures in the underlying distributed systems including hardware failures, whereas our failures were caused by defects in data-parallel programs.

5.1.2 Fix Characteristics Studies.

There were several studies on fixing patterns. Pan *et al.* [99] defined 27 bug-fixing patterns in Java software using syntax components involved in source-code changes. To fix bugs, Kim *et al.* [100] built *bug fix memories*, which were a project-specific knowledge base on bugs and the corresponding fixes. Some previous work [101, 102] also studied incorrect bug fixes. Again, our study about fixing patterns was specific to data-parallel programs, particularly for SCOPE jobs.

5.1.3 Debugging in Distributed Systems.

Olston *et al.* [103] provided a framework *Inspector-Gadget* for monitoring and debugging Pig Latin [44] jobs. Their interviews of programmers, which served as the motivation of their work, shared some common conclusions with our study, such as the needs for detecting exceptional data that violated certain data properties and needs for step-through debugging. However, their debugging support was different from ours. Instead of debugging on local machine in SCOPE, they provided a remote debugger on the failed commodity machine. Although remote debugging did not require downloading data to the local machine, it occupied shared resources on the commodity machine until the end of debugging process. Moreover, their debugger also had the root-cause problem described in our Finding 5 because their debugger enabled only partial-program debugging as we did. Some other work employed program-analysis techniques to help debugging in distributed systems. Liu *et al.* [104] used runtime checkers of program states to reveal how the program states turned bad. Taint analysis for data tracing [105, 106] was used to locate data that triggered failures.

5.2 TESTING AND ANALYSIS OF ACTOR SYSTEMS

5.2.1 Testing Actors

The most related work on testing Actor systems is dCUTE [53]. dCUTE differs from TAP in three aspects. First, dCUTE's goal is to achieve overall coverage while TAP aims at covering target code locations. They can be used to complement each other. Second, dCUTE performs *forward* concolic execution while TAP does *backwards* symbolic execution without a side-by-side concrete execution. Lastly, dCUTE handles only a subset of actor operations. For example, it assumes that all actors have been created before execution, and thus does not handle dynamic actor creation. , which is a commonly used operation to create actors dynamically. However, we provide a rigorous definition of the semantics of all actor operations in BSE.

BASSET [39] leverages the Java PathFinder model checker [107, 108] to systematically explore message schedules in an actor system. BASSET assumes that input messages are given, and aims at exploring as many message schedules as possible on the given input. It uses state merging and dynamic partial order reduction (DPOR) based on the happens-before relation to reduce the search space of message schedules. BITA [109] also explores possible message schedules for given input messages. However, BITA focuses its exploration on those schedules that are likely to expose bugs. It defines new *schedule coverage* criteria, and uses these criteria to guide the exploration to expose bugs. TRANSDPOR [110] proposes another DPOR technique that exploits the transitivity of the dependency relations between actors for schedule space reduction. TAP not only explores message schedules, but also generates message contents. These exploration techniques and space-reduction techniques can be integrated into TAP for more efficient test generation.

5.2.2 Targeted Test Generation

A number of targeted test generation techniques have been developed on sequential programs using both forward symbolic execution [54, 55, 56, 57, 58] and backward symbolic execution [36, 37, 38]. However, they cannot be directly applied to actor systems. Since an actor library often contains complex multi-threading and networking code, direct exploration of these actor library methods is impractical and the execution often fails to go across actors. Our work fills this gap by defining formal semantic models of actor operations in our analysis, and thus preventing our analysis from exploring the actor library.

5.2.3 Backward Symbolic Execution and Analysis.

BSE has been used previously in the literature to solve the line reachability problem for regular programs [54, 111, 36]. However, we are the first to support actor systems and to provide a rigorous formal definition of the BSE semantics. Ma et al. [54] proposes a solution to the line reachability problem by performing call-chain BSE to jump backward from one function call to another until reaching the entry point. Inside the functions, it performs forward symbolic execution. This is different from our work in that it doesn't target actor systems, and does not perform BSE in the intra-procedural analysis. Dinges et al. [36] combines BSE with concrete executions to solve the line reachability problem. Our work is different from [36] in that they do not give formal semantics for BSE and do not support distributed systems like actors. SNUGGLEBUG [111] uses backward symbolic *analysis* for computing inter-procedural weakest preconditions. The symbolic reasoning in their work is similar to ours except that their analysis works on all possible program paths to the target while our BSE aims at finding one feasible path. Moreover, our analysis is able to handle actor programs by defining analysis semantics on actor operations while their analysis works on only regular programs.

5.2.4 Feedback-Directed Test Generation

Previous research has proposed using information from previous executions as feedback to guide test generation. RANDOOP [112] uses execution feedback from previous tests to avoid generating redundant and illegal inputs. SEEKER [113] forms a feedback loop between static analysis and dynamic analysis to synthesize a method sequence that leads to a desired object state. Garg et al. [114] use the unsatisfiable cores from previous infeasible paths to generalize the reason for the infeasibility, and thus rule out more infeasible paths. We also use the unsatisfiable cores from infeasible paths, but we use them to guide BSE to efficiently find a feasible path.

5.2.5 Static Analysis of Actors

There has been previous work [115] on static analysis of actor programs to infer the ownership transfer of messages. This analysis works on individual actors (i.e., intra-actor), and does not model interactions between actors. Our MFG construction is a more complex whole-system analysis that requires modeling actor interactions. Note that MFG construction can serve as the foundation for other, more sophisticated whole-system analyses.

5.3 MODEL INFERENCE FOR CONCURRENT SYSTEMS

We review some of the research in inferring behavioral models for concurrent systems such as Petri net, MSC / MSG, CFSM, and UML Sequence Diagram.

One of the earliest models of concurrency is Petri nets [116]. The model was naturally defined as a graphical model, with places and tokens. Petri nets are not sufficiently powerful to represent actors as they lack recursion, but models of higher order Petri nets are. Two higher order variants of the Petri net model–Colored Petri nets and Predicate Transition Nets–are both equivalent to actors [117, 118]. In particular, one of the earliest visualization systems developed for actors was in terms of Predicate Transition Nets [118]. This visualization system directly translated actors into Predicate Transition Nets and allowed executions to be observed via the nets. The challenge in these systems is that there is no abstraction–thus making it infeasible to understand the behavior of even rather modestly complex systems in the presence of reconfiguration. Moreover, any kind of reconfiguration of actors makes observations difficult.

Another visualization system for actors was developed in [119]. This work also focused visualizing the dynamics of on-going executions. The system provides programming APIs to enable users to explicitly specify patterns of messages that could be observed in a Visualizer. The grammar of the Visualizer was based on visual connectors between actors. These connectors include merges and forks to represent coordination patterns. This enables the system to provide a degree of abstraction and avoid unnecessary details. However, specification of rules about what should be observed and how it should be abstracted are to the users; there is no program analysis to assist in the process. This means that users have to understand the interaction patterns of the code to begin with. On the other hand, the goal of our present research is to help program comprehension and debugging by extracting patterns of interaction that may be buried in the code.

Kumar *et al.* [74] proposed a dynamic mining framework for inferring MSG from execution traces of concurrent/distributed programs. They apply partial order mining techniques on a set of execution traces to identify frequently occurring interaction snippets and construct basic MSCs using these snippets. Then they use automata learning techniques to compose these basic MSCs into a MSG. Both their work and ours support the inference of behavioral models for concurrent systems based on message passing. However, there are also some fundamental differences. Kumar *et al.*'s inference method relies only on execution traces of the system and does not require the system code, while ours requires both. Hence, their approach towards model inference is to identify patterns and make generalization from a set of concrete execution traces, which is an under-approximation of the system. Our

approach is a combination of static analysis and dynamic analysis. The static analysis is used to construct an abstract model, which is an over-approximation of the system. The information from dynamic analysis is then used for refining the abstract model rather than generalizing concrete executions.

Kumar *et al.* [73] also proposed a dynamic specification mining framework for inferring class level system specifications. They proposed a new class level specification called symbolic MSC, which is more succinct than concrete MSC by grouping behaviorally similar processes. This work is incremental to their previous work on mining MSG. The addition in this work is the inference of symbolic MSCs from concrete MSCs. Thus this work is also fundamentally different from ours for the same aforementioned reasons.

Beschastnikh *et al.* [75] proposed CSight, a tool that infers CFSM of concurrent systems based on the system logs. Similar to the work [74, 73], CSight infers an initial model by generalizing concrete execution traces. However, it includes an additional step: it refines the initial model based on temporal properties mined from the system logs. CSight is similar to our work in that both approaches construct an initial model first, and then iteratively refine the initial model. Again, the construction of the initial model is different in the two approaches – CSight generalizes from an under-approximation, while our approach constructs an over-approximation from the system code. Hence, their model might miss some behaviors if they are not reflected in the concrete executions, while our model constructed by static analysis captures all possible behaviors. In addition, the refinement technique used by CSight is also different from ours. CSight employs a generic refinement technique called CEGAR [82], which uses counter examples to guide the refinement. On the other hand, we use a pattern-based refinement technique that is designed for refining our static analysis on actor systems. The advantage of a generic refinement technique is that it can be applied to refine any abstract model. The pattern-based technique may not work if the model matches none of the patterns. The advantage of the pattern-based technique is that it is very effective and can refine the model within a few iterations.

Oechsle *et al.* [120] proposed the JAVAVIS system to visualize Java programs with object diagrams and sequence diagrams. The system presents a direct visualization of the program dynamic behaviors without much abstraction. Briand *et al.* [121] proposed a methodology and an instrumentation infrastructure to construct UML sequence diagrams from dynamic executions. Rountev *et al.* [122] proposed a static analysis to construct UML sequence diagrams from the system code. They used an interprocedural dataflow analysis to map the interacting objects from the code to sequence diagram objects. These works are different from ours–they are either based on purely dynamic analysis or on purely static analysis. Our work combines static and dynamic analyses. In addition, the sequence diagrams constructed

in these works are simpler than actor specification diagrams: a sequence diagram does not have constructs such as loops and branches.

Efforts have been made in inferring dynamic invariants in distributed systems. Jiang *et al.* [123] proposed randomized algorithms to infer likely invariants in large-scale distributed systems. Their work aims for invariants on flow intensities which are internal performance measurements of how the system reacts to the volume of user requests. Our work infers different types of invariants such as temporal orders between actor events, and relationships between the numbers of actor events. Yabandeh *et al.* [124] developed a tool called *Avenger* to find *almost-invariants* in distributed systems, where *almost-invariants* are hypotheses that hold on all but a "few" execution traces. Our invariant inference method is similar to theirs except that we infer true invariants that hold on all execution traces instead of almost-invariants.

A number of mining techniques have been proposed to mine specifications in various formats that include frequent patterns and rules [125, 126, 127, 128, 129, 130], finite state machines [131, 132, 133, 134] from dynamic traces. These mining techniques mainly focus on specifications of sequential programs, while our inference method focuses on specifications of concurrent systems based on message passing.

Some previous research has focused on inferring system models from human-written specifications [76, 135, 136, 137, 138, 139, 140]. This approach requires significant manual efforts: specifications have to be written in the first place. In contrast, our inference method works directly on system code and execution traces without much manual efforts.

CHAPTER 6: CONCLUSION AND FUTURE WORK

In this chapter, we conclude our work in this dissertation and discuss some future research directions.

6.1 CONCLUSION

We have presented a comprehensive characteristic study on failures/fixes of production distributed data-parallel programs. We studied 250 failures of production SCOPE jobs, examining not only the failure types, failure sources, and fixes, but also current debugging practice. The major failure characteristic of data-parallel programs was that most of the failures (84.5%) were caused by defects in data processing rather than defects in code logic. The tremendous data volume and various dynamic data sources made data processing error-prone. In addition, there were limited major failure sources, with existing fix patterns for them, such as setting default values for null columns. We also revealed some interesting cases where the current SCOPE debugging tool did not work well and provided our suggestions for improvement.

We believe that our findings and implications provide valuable guidelines for future development of distributed data-processing programs, and also serve as motivations for future research on failure reduction and fixing in large-scale data-processing systems. In addition, our study results further indicates the importance and the necessity of system-level testing and analysis for better understanding the behaviors of concurrent systems and improving their reliability.

Motivated by this study, we have proposed a two-phased method for targeted test generation for actor systems based on BSE. In the first phase, our method constructs an MFG to capture the potential interactions between actors. Guided by the MFG, in the second phase, it starts BSE directly from the target to find a feasible path to the entry point of the actor system. To make our MFG analysis and BSE practical and scalable, we have provided high-level models for all actor operations and formally defined their semantics in our analysis to avoid analyzing the complex code in the actor library. To efficiently navigate the huge search space in BSE, we have proposed two heuristics and a feedback-directed search technique guide the path exploration in BSE.

We have implemented our method in Tap, and evaluated it on *Savina* and four open source projects. The evaluation results have shown that Tap is effective in covering target code location in actor systems. It achieves an overall 78% target coverage on 1000 randomly

selected targets. The results also demonstrate that TAP is effective in detecting concurrency bugs. It detects six distinct crashing concurrency bugs that are not previously reported in our subjects.

To help developers understand and reason about actor systems, we have proposed an inference method that infers actor specification diagrams, which rigorously depict the global behaviors of actor systems. Our inference method is a combination of static analysis and dynamic analysis. It first statically analyzes the system code to build an abstract specification diagram, which captures not only all potentially message flows, but also the temporal orders between the message events. Then our method infers dynamic likely invariants from the execution traces to further refine the diagram from static analysis. The invariants can help infer the number of actors and messages as well as discover additional orders between events enforced through coordination constraints in the program. We implement the inference method in a tool ASpec, and evaluate ASpec on the *Savina* benchmarks as well as two real-world protocols TCP and SIP. The evaluation results show that ASpec is capable of inferring actor specification diagrams with high accuracy.

There are many potential applications of the inferred specification diagrams. A common use case is for developers to understand the global behavior of a system at a high level, especially the interactions between actors. Many scenarios in practice require understanding the system behavior, for example, when maintaining and working with legacy code. It may also be used to detect bugs in the implementation. Developers can check the specification diagram inferred from an actual implementation to see whether the specified behavior is intended. If not, there might be a bug in the implementation. Finally, the specification diagram may also be used as the model in model-based testing and model checking.

6.2 FUTURE WORK

Efficient Exploration of Message Schedules. Previous work has used partial order reduction to help efficiently explore message schedules in actor systems. Various partial order reduction techniques have been developed and studied in terms of their efficiency for space reduction. The fundamental idea of partial order reduction is that if multiple messages are independent with each other, then we only need to explore one possible schedule between these messages, because all the other schedules would result in the same behavior.

We believe that there is still room to improve the efficiency of existing partial order reduction techniques by refining the definition of dependency between messages. In the previous work, two messages are considered dependent if they are received by the same actor. This is a quite conservative condition on dependency because there are many cases

in which these two messages do not depend on each other. For instance, both messages do not change the local states of the actor, or they change different parts of the local states. Consequently, partial order reduction techniques based on this dependency definition may miss many reduction opportunities. We plan to use program analysis to further reason about the dependency between messages. In particular, we check whether the states affected by these two messages overlap or not. We also plan to integrate the partial order reduction techniques into our test generation tools to improve the efficiency of path exploration.

Forward Execution and Backward Execution. There have been targeted test generation techniques using forward symbolic execution and those using backward symbolic execution. Theoretically, it is unclear which method is more efficient. We use backward symbolic execution because it naturally explores only the paths that may lead to the target. However, with the guidance of a global model of the actor system, forward symbolic execution can also avoid exploring irrelevant paths. We are interested to compare these two methods empirically. We plan to apply both methods to an extensive set of subjects, and compare their effectiveness in terms of target coverage and bug detection.

Inferring Invariants of Actor System. We have demonstrated the effectiveness of using dynamic invariants to refine the actor specification diagrams. The invariants we inferred are basic invariants in the form of linear relationships of the number of messages or the number of actors, as well as temporal orders between actor events. We are interested to discover more advanced and complex invariants, which describes the coordination between a group of actors. Such invariants would be more useful for helping developers understand and reason about the actor systems.

We believe one key factor to enable the inference of such invariants is to understand the synchronization between actors. In the Actor model, the synchronization is often enforced through the local state change of one or more actors. This observation indicates that the invariant templates we provide for dynamic invariant inference should be able to express such synchronization between a group of actors. We plan to investigate interesting invariants on synchronization from real-world concurrent and distributed protocols, and try to identify common forms of such invariants for hypothesis generation. With a richer set of hypotheses, we would be able to discover more complex invariants.

Verification on Distributed Protocols. We are also interested in formal verification of distributed protocols. Real-world implementations of distributed systems are often very complex, posing significant challenges to verifying them in practice. For example, the actual production implementations of the Paxos protocol [141] are known to be much more complex than the description of the protocol in the original paper [142]. Recently, a new approach [143] has been proposed to verify the model of the actual implementation of the core

protocol in the system. The model itself is an executable program. By verifying the model, it also increases the confidence on the correctness of the underlying implementation. We are interested to adopt this methodology and use the Actor model for verification. In particular, the models in the previous work are multi-thread programs. We plan to model those distributed protocols using the Actor model, which we believe is a more natural programming model for these protocols. There are likely new challenges for verification techniques on the Actor model.

Usefulness of Behavioral Models. We are interested to see how useful the actor specification diagram is to help developers understand the system. Particularly, we plan to conduct user studies to evaluate its usefulness in terms of helping developers detect bugs in the system. We can carry out a controlled experiment, where we ask two groups of people to identify bugs in the implementations of actor systems, and compare how effective they are to find bugs. For one group, we provide a faulty implementation with manually injected bugs, along with the correct description and test cases of the system, For the other group, besides the implementation, description, and test cases, we also provide the actor specification diagram inferred from the faulty implementation. Then we compare the two groups by the number of bugs detected and the time taken to detect these bugs.

REFERENCES

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA, USA: MIT Press, 1986.

[2] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *Journal of Functional Programming*, vol. 7, no. 1, pp. 1–72, 1997.

[3] G. Agha, "Concurrent Object-oriented Programming," *Commun. ACM*, vol. 33, no. 9, pp. 125–141, Sep. 1990.

[4] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight remote procedure call," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 5, pp. 102–113, 1989.

[5] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight remote procedure call," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, ser. SOSP '89. New York, NY, USA: ACM, 1989. [Online]. Available: http://doi.acm.org/10.1145/74850.74861 pp. 102–113.

[6] A. M. Goldsmith, D. B. Goldsmith, and C. E. Pettus, "Object-oriented remote procedure call networking system," Feb. 13 1996, uS Patent 5,491,800.

[7] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[9] C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will, and G. Agha, "Rosette: An object-oriented concurrent systems architecture," *SIGPLAN Notices*, vol. 24, no. 4, pp. 91–93, 1989. [Online]. Available: https://doi.org/10.1145/67387.67410

[10] C. R. Houck and G. Agha, "HAL: A high-level actor language and its distributed implementation," in *Proceedings of the 1992 International Conference on Parallel Processing, University of Michigan, An Arbor, Michigan, USA, August 17-21, 1992. Volume II: Software*, K. G. Shin, Ed. CRC Press, 1992, pp. 158–165.

[11] W. Kim and G. Agha, "Efficient support of location transparency in concurrent object-oriented programming languages," in *Proceedings Supercomputing '95, San Diego, CA, USA, December 4-8, 1995*, S. Karin, Ed. ACM, 1995. [Online]. Available: https://doi.org/10.1145/224170.224297 p. 39.

[12] J. Armstrong, *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[13] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with salsa," *SIGPLAN Not.*, vol. 36, no. 12, pp. 20–34, Dec. 2001.

[14] *The E language*, http://www.erights.org/elang/.

[15] E. A. Lee and I. John, "Overview of the ptolemy project," 1999.

[16] "Axum programming language." [Online]. Available: https://msdn.microsoft.com/en-us/devlabs/ dd795202.aspx

[17] *Erlang Introduction*, http://erlang.org/faq/introduction.html.

[18] *Scala Akka*, https://doc.akka.io/docs/akka/current/index-actors.html?language=scala.

[19] *LiftActor framework*, http://liftweb.net/api/25/api/net/liftweb/actor/ LiftActor.html.

[20] *Java Akka*, https://doc.akka.io/docs/akka/current/actors.html?language=java.

[21] M. Rettig, "jetlang: Message based concurrency for java," *URL http://code. google. com/p/jetlang*, 2014.

[22] M. Astley, "The actor foundry: A java-based actor programming environment," *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.

[23] *The GPars framework*, http://www.gpars.org/guide/.

[24] D. G. Kafura and K. H. Lee, "Act++: Building a concurrent c++ with actors," Blacksburg, VA, USA, Tech. Rep., 1989.

[25] J.-P. Briot, "Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment." in *ECOOP*, vol. 89, 1989, pp. 109–130.

[26] C. Tismer, "Continuations and stackless python," in *Proceedings of the 8th international python conference*, vol. 1, 2000.

[27] J. Ayres and S. Eisenbach, "Stage: Python with actors," in *Multicore Software Engineering, 2009. IWMSE'09. ICSE Workshop on*. IEEE, 2009, pp. 25–32.

[28] *Microsoft Asynchronous Agents Library*, https://msdn.microsoft.com/en-us/ library/dd492627.aspx.

[29] M. Rettig, *retlang: Message based concurrency in .NET*, 2010.

[30] *Orleans*, https://dotnet.github.io/orleans/index.html.

[31] R. Chaiken, B. Jenkins, P. ke Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and efficient parallel processing of massive data sets," *PVLDB*, vol. 1, no. 2, pp. 1265–1276, 2008.

[32] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007, pp. 59–72.

[33] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *CCGRID*, 2010, pp. 94–103.

[34] Apache, "Hadoop," http://hadoop.apache.org/. [Online]. Available: http://hadoop.apache.org/

[35] Yahoo!, "M45 supercomputing project," http://research.yahoo.com/ node/1884. [Online]. Available: http://research.yahoo.com/ node/1884

[36] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 31–36.

[37] O. Olivo, I. Dillig, and C. Lin, "Detecting and exploiting second order denial-of-service vulnerabilities in web applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 616–628.

[38] F. Charreteur and A. Gotlieb, "Constraint-based test input generation for java bytecode," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 131–140.

[39] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha, "A Framework for State-Space Exploration of Java-Based Actor Programs," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09, Washington, DC, USA, 2009, pp. 468–479.

[40] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ACM Sigplan Notices*, vol. 43, no. 3. ACM, 2008, pp. 329–339.

[41] S. Imam and V. Sarkar, "Savina-an actor benchmark suite," in *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE*, 2014.

[42] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proc. ICSE*, 1999, pp. 213–224.

[43] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin, and T. Xie, "A characteristic study on failures of production distributed data-parallel programs," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 963–972.

[44] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A not-so-foreign language for data processing," in *SIGMOD*, 2008, pp. 1099–1110.

[45] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Li, and R. Murthy, "Hive – a petabyte scale data warehouse using Hadoop," in *ICDE*, 2010, pp. 996–1005.

[46] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," in *PLDI*, 2010, pp. 363–375.

[47] N. Tillmann and J. de Halleux, "Pex-white box test generation for .NET," in *TAP*, 2008, pp. 134–153.

[48] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. PLDI*, 2005, pp. 213–223.

[49] K. Sen, D. Marinov, and G. Agha, "CUTE: a Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, New York, NY, USA, 2005, pp. 263–272.

[50] Microsoft, "Nullable Types," http://msdn.microsoft.com/en-us/library/ 1t3y8s4s(v=vs.80).aspx. [Online]. Available: http://msdn.microsoft.com/en-us/library/ 1t3y8s4s(v=vs.80).aspx

[51] Microsoft, "FxCop," http://msdn.microsoft.com/en-us/library/ bb429476(v=vs.80).aspx. [Online]. Available: http://msdn.microsoft.com/en-us/library/ bb429476(v=vs.80).aspx

[52] Microsoft, "Phoenix Compiler," http://research.microsoft.com/en-us/collaboration/ focus/cs/phoenix.aspx. [Online]. Available: http://research.microsoft.com/en-us/collaboration/ focus/cs/phoenix.aspx

[53] K. Sen and G. Agha, "Automated systematic testing of open distributed programs," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2006, pp. 339–356.

[54] K.-K. Ma, K. Yit Phang, J. Foster, and M. Hicks, "Directed symbolic execution," *Static Analysis*, pp. 95–111, 2011.

[55] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations." in *USENIX Security Symposium*, 2013, pp. 49–64.

[56] J. Feist, L. Mounier, and M.-L. Potet, "Guided dynamic symbolic execution using subgraph control-flow information," in *International Conference on Software Engineering and Formal Methods*. Springer, 2016, pp. 76–81.

[57] P. D. Marinescu and C. Cadar, "Katch: high-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 235–245.

[58] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 12–22.

[59] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial intelligence*, vol. 8, no. 3, pp. 323–364, 1977.

[60] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: a minimal core calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 3, pp. 396–450, 2001.

[61] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 17–30.

[62] M. Might, Y. Smaragdakis, and D. Van Horn, "Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 305–315.

[63] J. Schäfer and A. Poetzsch-Heffter, "JCoBox: Generalizing Active Objects to Concurrent Components," in *ECOOP 2010 – Object-Oriented Programming*, T. D'Hondt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 275–299.

[64] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 1, pp. 1–41, 2005.

[65] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg, 2008, pp. 337–340.

[66] *Wala*, http://wala.sourceforge.net.

[67] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Caf-the C++ actor framework for scalable and resource-efficient applications," in *Proceedings of the 4th International Workshop on Programming based on Actors Agents and Decentralized Control*. ACM, 2014, pp. 15–28.

[68] A. Rosà, L. Y. Chen, and W. Binder, "Profiling actor utilization and communication in Akka," in *Proceedings of the 15th International Workshop on Erlang*. ACM, 2016, pp. 24–32.

[69] G. Upadhyaya and H. Rajan, "Effectively mapping linguistic abstractions for message-passing concurrency to threads on the Java virtual machine," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 840–859, 2015.

[70] S. Tasharofi, P. Dinges, and R. E. Johnson, "Why do scala developers mix the actor model with other concurrency models?" in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 302–326.

[71] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of uml sequence diagrams for distributed java software." *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 642–663, 2006.

[72] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts." in *ICSE*, C. Ghezzi, M. Jazayeri, and A. L. Wolf, Eds. ACM, 2000, pp. 304–313.

[73] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, "Inferring class level specifications for distributed systems," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 914–924.

[74] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo, "Mining message sequence graphs," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 91–100.

[75] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 468–479.

[76] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker, "Learning communicating automata from mscs," *IEEE Transactions on Software Engineering*, vol. 36, no. 3, pp. 390–408, 2010.

[77] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Reading, Massachusetts etc.: Addison-Wesley Professional, September 1998.

[78] I. T. S. Sector, "ITU-T Recommendation Z. 120," *Message Sequence Charts (MSC96)*, 1996.

[79] A. Muscholl and D. Peled, "Message sequence graphs and decision problems on mazurkiewicz traces," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1999, pp. 81–91.

[80] J. G. Henriksen, M. Mukund, K. N. Kumar, and P. Thiagarajan, "On message sequence graphs and finitely generated regular msc languages," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2000, pp. 675–686.

[81] D. Brand and P. Zafiropulo, "On communicating finite-state machines," *Journal of the ACM (JACM)*, vol. 30, no. 2, pp. 323–342, 1983.

[82] D. Kroening, A. Groce, and E. M. Clarke, "Counterexample guided abstraction refinement via program execution," in *Proc. ICFEM*, 2004, pp. 224–238.

[83] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha, "Actively learning to verify safety for fifo automata," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 2004, pp. 494–505.

[84] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha, "Learning to verify safety properties," in *International Conference on Formal Engineering Methods*. Springer, 2004, pp. 274–289.

[85] S. F. Smith and C. L. Talcott, "Specification diagrams for actor systems," *Higher-Order and Symbolic Computation*, vol. 15, no. 4, pp. 301–348, 2002.

[86] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," *ACM SIGPLAN Notices*, vol. 43, no. 1, pp. 273–284, 2008.

[87] R. Neykova and N. Yoshida, "Multiparty session actors," in *International Conference on Coordination Languages and Models*. Springer, 2014, pp. 131–146.

[88] M. Charalambides, P. Dinges, and G. Agha, "Parameterized, concurrent session types for asynchronous multi-actor interactions," *Science of Computer Programming*, November 2015.

[89] S. Li, F. Hariri, and G. Agha, "Targeted Test Generation for Actor Systems," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), T. Millstein, Ed., vol. 109. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9213 pp. 8:1–8:31.

[90] "RFC 793 Transmission Control Protocol," https://tools.ietf.org/html/rfc793.

[91] "RFC 3261 SIP: Session Initiation Protocol," https://tools.ietf.org/html/rfc3261.

[92] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *SOSP*, 2001, pp. 73–88.

[93] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang, "Characterization of Linux kernel behavior under errors," in *DSN*, 2003, pp. 459–468.

[94] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *DSN*, 2000, pp. 97–106.

[95] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008, pp. 329–339.

[96] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 249–265.

[97] S. A. Asadollah, D. Sundmark, S. Eldh, H. Hansson, and E. P. Enoiu, "A study of concurrency bugs in an open source software," in *IFIP International Conference on Open Source Systems*. Springer, 2016, pp. 16–31.

[98] H. Jin, K. Qiao, X.-H. Sun, and Y. Li, "Performance under failures of MapReduce applications," in *CCGRID*, 2011, pp. 608–609.

[99] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, 2009.

[100] S. Kim, K. Pan, and E. E. J. Whitehead, Jr., "Memories of bug fixes," in *FSE*, 2006, pp. 35–45.

[101] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *ESEC/FSE*, 2011, pp. 26–36.

[102] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR*, 2005, pp. 1–5.

[103] C. Olston and B. Reed, "Inspector Gadget: A framework for custom monitoring and debugging of distributed dataflows," in *SIGMOD*, 2012, pp. 1221–1224.

[104] X. Liu, W. Lin, A. Pan, and Z. Zhang, "Wids checker: combating bugs in distributed systems," in *NSDI*, 2007, pp. 19–19.

[105] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: a pervasive network tracing framework," in *NSDI*, 2007, pp. 20–20.

[106] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," in *SOSP*, 2003, pp. 74–89.

[107] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[108] *Java PathFinder*, http://javapathfinder.sourceforge.

[109] S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson, "Bita: Coverage-guided, automatic testing of actor programs," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 114–124.

[110] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha, "TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs," in *Formal Techniques for Distributed Systems*. Springer, 2012, pp. 219–234.

[111] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 363–374, 2009.

[112] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, Washington, DC, USA, 2007, pp. 75–84.

[113] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," in *Proc. OOPSLA*, 2011, pp. 189–206.

[114] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for C/C++ using concolic execution," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 132–141.

[115] S. Negara, R. K. Karmani, and G. Agha, "Inferring ownership transfer for efficient message passing," in *ACM SIGPLAN Notices*, vol. 46, no. 8. ACM, 2011, pp. 81–90.

[116] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9, no. 3, pp. 223–252, 1977.

[117] Y. Sami and G. Vidal-Naquet, "Formalisation of the behavior of actors by colored petri nets and some applications," in *PARLE '91: Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 10-13, 1991, Proceedings*, ser. Lecture Notes in Computer Science, E. H. L. Aarts, J. van Leeuwen, and M. Rem, Eds., vol. 506. Springer, 1991. [Online]. Available: https://doi.org/10.1007/3-540-54152-7_61 pp. 110–127.

[118] S. Miriyala, G. Agha, and Y. Sami, "Visualizing actor programs using predicate transition nets," *J. Vis. Lang. Comput.*, vol. 3, no. 2, pp. 195–220, 1992. [Online]. Available: https://doi.org/10.1016/1045-926X(92)90015-E

[119] M. Astley and G. Agha, "A visualization model for concurrent systems," *Inf. Sci.*, vol. 93, no. 1, pp. 107–131, 1996. [Online]. Available: https://doi.org/10.1016/0020-0255(96)00063-1

[120] R. Oechsle and T. Schmitt, "Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi)," in *Software Visualization*, S. Diehl, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 176–190.

[121] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of uml sequence diagrams for distributed java software," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 642–663, Sep. 2006.

[122] A. Rountev and B. H. Connell, "Object naming analysis for reverse-engineered sequence diagrams," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, May 2005, pp. 254–263.

[123] G. Jiang, H. Chen, and K. Yoshihira, "Efficient and scalable algorithms for inferring likely invariants in distributed systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 11, pp. 1508–1523, Nov 2007.

[124] M. Yabandeh, A. Anand, M. Canini, and D. Kostic, "Finding almost-invariants in distributed systems," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, Oct 2011, pp. 177–182.

[125] D. Lo and S.-C. Khoo, "Smartic: Towards building an accurate, robust and scalable specification miner," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 265–275.

[126] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 460–469.

[127] D. Lo, S.-C. Khoo, and C. Liu, "Mining past-time temporal rules from execution traces," in *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, ser. WODA '08. New York, NY, USA: ACM, 2008, pp. 50–56.

[128] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, June 2006, pp. 84–88.

[129] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal api rules from imperfect traces," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 282–291.

[130] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: From usage scenarios to specifications," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 25–34.

[131] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 501–510.

[132] L. Mariani and M. Pezze, "Behavior capture and test: automated analysis of component integration," in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, June 2005, pp. 292–301.

[133] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin, "Reverse engineering state machines by interactive grammar inference," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, Oct 2007, pp. 209–218.

[134] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 371–382.

[135] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Learning operational requirements from goal models," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 265–275.

[136] C. Damas, B. Lambeau, and A. van Lamsweerde, "Scenarios, goals, and state machines: A win-win partnership for model synthesis," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 197–207.

[137] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel, "Validation of contracts using enabledness preserving finite state abstractions," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 452–462.

[138] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-11. New York, NY, USA: ACM, 2003, pp. 257–266.

[139] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic, "Synthesizing partial component-level behavior models from system specifications," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 305–314.

[140] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 314–323.

[141] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007, pp. 398–407.

[142] L. Lamport et al., "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[143] E. Pek, P. Garg, M. R. Rahman, I. Gupta, and P. Madhusudan, "Certified program models for eventual consistency."